Summer 8-1-2014

IMPROVING PREFERENCE RECOMMENDATION AND CUSTOMIZATION IN REAL WORLD HIGHLY CONFIGURABLE SOFTWARE SYSTEMS

Dongpu Jin
University of Nebraska-Lincoln, djin@cse.unl.edu

IMPROVING PREFERENCE RECOMMENDATION AND CUSTOMIZATION IN REAL WORLD HIGHLY CONFIGURABLE SOFTWARE SYSTEMS

by

Dongpu Jin

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Myra B. Cohen

Lincoln, Nebraska

August, 2014

IMPROVING PREFERENCE RECOMMENDATION AND CUSTOMIZATION IN REAL WORLD HIGHLY CONFIGURABLE SOFTWARE SYSTEMS

Dongpu Jin, M.S.

University of Nebraska, 2014

Adviser: Myra B. Cohen

Highly configurable software systems, such as web browsers or office applications, may have a large number of preferences that the user can customize. When faced with the task of trying to identify which configuration option should be modified to change a particular system behavior, the user, tester or debugger may have to search through hundreds or thousands of options, and documentation may be scarce. Simple pattern matching utilities exist, but these searches are sensitive to using the right keyword. Static analysis may help, but will require access to source code. Alternatively a user may ask questions on help forums, but this can takes hours, days or even weeks to obtain a solution.

In this thesis we begin by analyzing two open-source and one industrial application to understand the complexity of their configuration subsystems. We find that all applications are multi-lingual, that there are multiple access points and methods to modify configurations, and only a subset of preferences are provided through the use of a menu option. These results suggest the need for new recommendation and customization approaches. We then present PrefFinder, an automated framework that uses natural language processing and information retrieval to search for preferences. The input is a query in natural language and the result is a rank ordered list of the potential options, and an update mechanism that allows the user to directly change the found preference at run time. We instantiate PrefFinder as a plugin for Firefox and evaluate several variants of our parsing algorithms to improve matches in this context. On 100 queries obtained from an online forum, we

determine that using a backward search during word splitting, combined with a synonym database, achieves the best retrieval results. The correct configuration option is found 50 percent of the time within the top 20 choices, and 73 percent of the time overall. In a comparison against a standard web search, we show that PrefFinder is competitive in finding the answer, but at a potentially lower cost.

ACKNOWLEDGMENTS

Firstly, I would like to give my thanks to my adviser Dr. Myra Cohen. I really enjoyed working with her throughout the two-year master program. Her guidance played an important role in helping my research, thesis, coursework, study, and internship. By working under her supervision, I learned not only tremendous amount of major specific knowledge, but also very essential soft skills such as communication, academic writing, interpersonal, and the rigorous academic attitudes. I really appreciate her patient and kindly personality, which makes the two-year research experience really motivated, productive, and enjoyable.

Secondly, I would like to give my appreciations to my parents, whose endless love and support have become the strongest source of my energy and helped me persesvere and continuously move forward. I believe the sacrifice of not being together with my family would eventually pay off as I successfully complete my master degree.

Last but not the least, I would like to thank my committee members who spending time reading this thesis, attending my defense, and providing valuable feedback. I would also like to give my thanks to the professors, students, researchers, and staff in the e2 lab and computer science department, who are all wonderful individuals that provided me with tremendous guidance and help, which made this a really wonderful journey.

This work was supported in part by the National Science Foundation grant #CCF1161767, CNS #1205472 and the Air Force Office of Scientific Research award #FA9550-10-1-0406.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many software systems are *highly-configurable*, allowing the user to customize an individual instance of the program while retaining a core set of functionality. Users can customize a program's behavior by specifying option settings for a large number of preferences (often in the hundreds or thousands). During development, maintenance and testing, engineers will also manipulate the preferences to ensure that correct behavior occurs under a wide range of user profiles. This customizability provides benefit to the end-user, however, it also introduces many challenges during testing and/or debugging, because configurability complicates the process of finding and/or reproducing the failure. Research has shown that different instances of a highly-configurable system will behave differently while running under the same set of test cases [8, 39, 55]. For instance, in the work of Qu et al. [39], as many as 80% of the faults had the potential to go undetected if tested under certain configurations. Therefore, configuration-aware testing techniques have been proposed, to systematically explore the configuration space [39, 55]. During debugging, configurations are also important. Knowing the exact configuration instance that a user was in when the failure occurred can help with reproducibility. Bettenburg et al. [2] found that there is a strong mismatch in bug reports between what developers need to reproduce and fix a bug,

and that which is provided by users. Other studies have also shown that bug reports lack information needed for bug reproduction [4]. Although there has been some work aimed at reproducing field failures such as that of Jin and Orso and Clause and Orso [5, 24], it does not explicitly consider the configuration at the time of failure.

Given the complexity of today's software systems, determining the configuration space may not be a trivial task. For instance, in the industrial system studied in Qu et al. [42], they reported that there are more than 500 configuration options that their users can modify. Firefox, the open source web browser has over 1,900 configuration options available to a user. The space of possible unique configurations grows exponentially with the number of configuration options (also called preferences in this work), therefore we can only evaluate a representative sample of all possible configurations. Research in testing and maintenance of configurable software has focused on ways to sample this large configuration space for testing [14, 51], or to prioritize these samples to improve efficiency during maintenance [40]. Rabkin and Katz highlight the lack of documentation on which configuration options exist, and on what the valid value domains are for each of those options. [44]. They have developed a static analysis technique that reverse engineers the configuration options from code [44]. In follow on work, they have proposed methods to diagnose possible errors found during configurability [43], and Xiong et al. have developed a symbolic technique to provide fixes when configuration constraints are violated [54]. Zhang and Ernst have developed another analysis to identify which configuration option causes a failure [57] or has caused the system behavior to change in an undesirable way [58] due to evolution.

However, if users and developers (or testers) want to interact with a configurable system, during use or maintenance, these systems may be lacking. They may know of a desired behavior, or be familiar only with a descriptive menu name for a specific configuration option, but there is no way to query most systems to map these *human readable* preferences to code-level names. For instance, if a developer knows that a preference on the menu option

is called *Always show the tab bar*, they may not be able to quickly determine what the real preference name is, but that is necessary if one wants to set this automatically via a utility function or by directly modifying a preference file. In this case, the preference is called *browser.tabs.autoHide*. While this particular option can also be set by the menu, making it seem trivial, some can only be manipulated in other ways.

Consider for example, the *browser.backspace_action*, found in Firefox. This allows the user to control how the browser will behave when using the backspace key. There is no menu setting for this option, yet a sophisticated user and/or a developer may want to modify its settings. Luckily in this scenario, Firefox provides a utility, `about:config` (see Figure 1.1), that uses a regular expression search to find matching options. If the user happens to search for *backspace* using this utility, they will find this option and be able to modify its settings. However, if instead they decide to search for *back space* or *spacebar*, nothing will be returned. Browsing through all options in `about:config` will be difficult, since there are almost 1900 options in the current version. If the user is working on a system like LibreOffice, they have a directory with many subdirectories to search in order to find the possible options. Instead, we would like to use natural language to *describe* the behavior and find the options another way.

In this thesis we first uncover and quantify the extent to which these problems exist on an industrial scale and then address the configurability problems through an automated preference finder suggestion framework, that we call *PrefFinder*. To achieve the first goal, we empirically examine several large highly-configurable applications to understand the implications for testing and debugging in practice. We study one industrial application and two widely used open source applications. We quantify the size of the configuration space and evaluate where and if the ground truth for the configuration model exists. We also examine how a user, tester or maintenance engineer can manipulate the configuration options. Finally, we examine the runtime factors involved in capturing the current

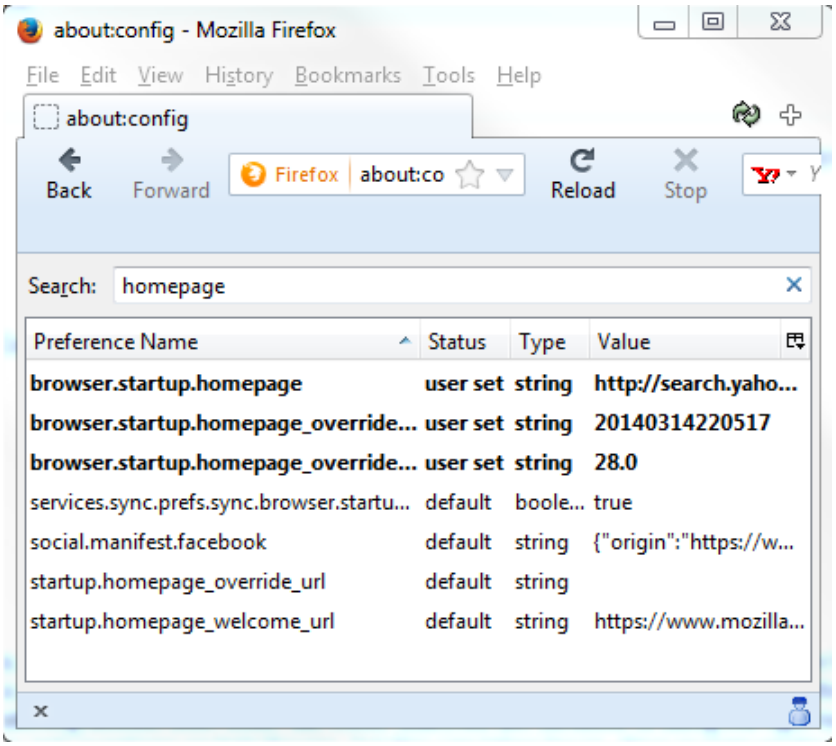

Figure 1.1: Firefox `about:config` utility.

configuration space. Our study shows, somewhat surprisingly, that both the industrial and open source applications have elements of configurability in common, which leads to a set of lessons learned and a roadmap for developing configuration-aware testing and debugging tools. We see this study as a way to share with practitioners the issues configurability brings, and a springboard to accurate and usable configuration-aware testing and debugging techniques.

To address the second goal of preference recommendation and customization of real world highly configurable software systems, we built PrefFinder, an automated framework that uses natural language processing and information retrieval to help locate the desired preferences. The input is a query in natural language. PrefFinder first parses both the preferences and the user query, informed by dictionaries and lexical databases. The queries and preferences are then matched and ranked and returned to the user along with current

values and brief preference descriptions if exists. We have built a prototype of PrefFinder for Firefox browser as a plugin. In experiments on a set of 100 real user queries on Firefox browser, we show that PrefFinder finds the correct preference 73 percent of the time overall and 50 percent of the time within the top 20 choices.

The contributions of this thesis are:

1. An abstraction of the general structure of configuration manipulation in modern software systems;

2. A case study that quantifies the complexity of three modern highly configurable software systems and a set of lessons learned that will help practitioners to better understand and control configuration instances for software engineering tasks such as testing and debugging;

3. PrefFinder: A framework to provide interactive querying of configurable options in natural language;

4. A prototype PrefFinder implementation for the Firefox web browser; and

5. A case study on 100 user queries evaluating the effectiveness of our various splitting algorithms and PrefFinder. itself

The rest of this thesis is structured as follows. In the next chapter we present the background and related work. In Chapter 3 we present an analysis of configuration on real software systems. We then present PrefFinder in Chapter 4. We conclude and present future directions in Chapter 5.

Chapter 2

Background and Related Work

We begin with a discussion of configurability and then present background on natural language processing. We end with related work.

2.1 Background

2.1.1 Configurable Software Systems

A *configurable system* is a software system with a core set of functionality and a set of variable features which are defined by a set of *configuration options* (or *preferences*). Changes to the *value* of a preference changes the program's behavior in some way. For instance, Firefox, a popular web browser, is a highly-configurable system and one that we use to motivate some of the problems that we have encountered. In Firefox, an example configuration option that can be set via the option menu is called *Warn me when closing multiple tabs*. This is a Boolean configuration with two values, $\{$*true, false*$\}$. Its default value is set to *true* which means that if you try to close a window when multiple tabs are open, you will get a warning asking if you want to close all of the tabs. If you uncheck this on the menu (set it to false) it will prevent a warning from being produced and immediately close

the window. The actual preference name for this (found in the preference file) is called *browser.tabs.warnOnClose*. There is another closely related preference in the preference files called *browser.tabs.warnOnCloseOtherTabs* which is set to true by default, but has no menu counterpart. When testing the system, or when a failure occurs, we need to have information about the values that were selected for each of these configuration options, something that may not be obvious by examining just the menu alone.

We assume an idealized use case for testing and debugging as shown in Figure 2.1. In this scenario we have three entities that interact with the configurable system. The end-user can modify configurations and will send bug reports to (and possibly read reports from) customer support. As can be seen in the figure, he or she may use the menu, or they can directly write to configuration files. A set of configuration-aware techniques and tools sit between the application and the tester and maintenance engineer, which feed information about configurations back to the bug reporting/customer support system. The challenge is to enable these configuration-aware techniques. We have identified three important requirements. We need

1. **a Model** of the possible configuration space. In order to sample the configuration space for testing or debugging, the configuration model needs to be known.
2. **to Know the Mapping** of the configuration space to programmatic elements. This is required in order to understand the impact a configurable item can have, and to automate the modification of configurations for testing and bug reproduction.
3. **an Accurate Configuration Snapshot** to provide the full state of the application when a bug is encountered.

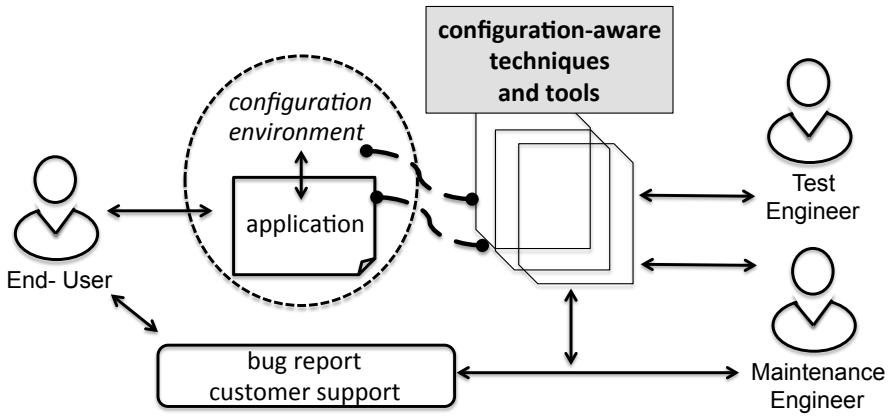

Figure 2.1: Configuration-Aware Testing and Debugging: Expected Use Case

We examine each of these requirements in relation to the existing work. Configuration-aware testing techniques [7, 14, 39] propose various methods to sample and prioritize the configuration space for testing, but all of this work assumes that the **configuration model** is known (or is somehow extracted from the code). Based on our informal examination of systems like Firefox, we do not believe that this can be easily achieved. First, we have discovered that the configuration control is not found within a single location of the code or in specific external files. In fact, most of the systems we have studied have a multi-tiered layout of how configurations are defined and accessed and this can be done both offline and at run time. Figure 2.2 shows a schema that seems to cover most of the systems we have studied. First, there is a *static view* of the system (labeled #1). This includes any existing user manuals, web pages, etc. that contain documentation on the possible configuration options and their values. This often is incomplete or out of date. The second static element is the source code itself. This contains the *ground truth*, but source code may not be available to everyone who wants and needs to understand the configuration model. Moreover, as we shall see, using this to extract the full configuration space is non-trivial.

When controlling what configurations are set, there are usually external mechanisms

(#2 in Figure 2.2) such as preference files or databases. These can often be accessed independently of the program (even while it is running) and therefore may or may not contain the current state of the configurations. We have also seen that these may not contain the ground truth of the configuration space.

Finally, as is shown in #3, there are usually some runtime access mechanisms that connect to the internal data structures (or database). For instance, most programs have a menu system that allows the user to set preferences, but in the systems we have studied this accounts for only a subset of the full set of configurations. Other specialty tools exist such as the **about:config** mechanism of Firefox, that allows one to pull up a web page where configurations can be modified dynamically. Again, these may not show the complete set of configuration options that are available. There may also be an API to allow programmatic access to an internal memory structure (such as the hash table in Firefox). This should be the ground truth of what preferences are set at any point in time, but it will not contain the hidden preferences.

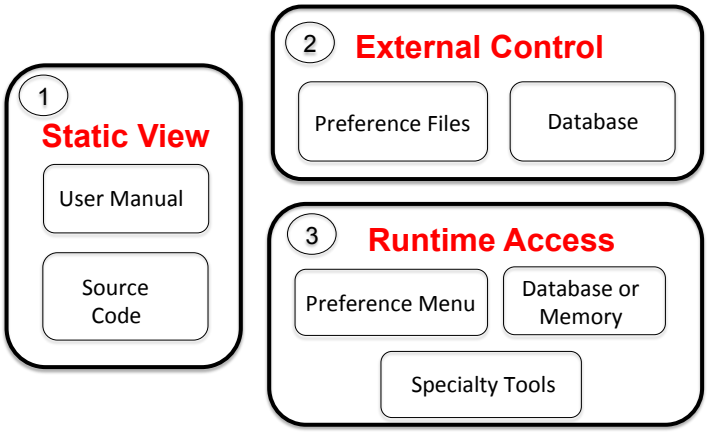

Figure 2.2: General View of Configuration Layers

Suppose instead of using the menus or preference files, we want to extract the preferences from the code itself, which also helps to build a **mapping** between the configuration

space and code. Rabkin et al. [45, 46] presented techniques to statically analyze Java programs with JChord. However, upon studying their work in more detail, we find that it does not directly apply to a system like Firefox. First, it assumes a single programming language (Java); second, they assume that all of the preference manipulation code exists as (**name, value**) pairs and is found in a single class; and finally, they assume that configuration manipulation methods start with *get* or *set*.

As shall see, these assumptions do not hold for any of the applications we studied. For instance, there are cases in Firefox where the preference code includes JavaScript and other languages such as the markup language XUL. We see instances where the Javascript API is able to query and update a preference, however, it uses the XUL code as a reference to the given preference name (binding it to a user interface element). We also see preference code that is not using the (name, value) pair mechanism but instead uses references, macros, or member fields to refer to the preference name. Another issue that we have encountered is that the API method names of Firefox do not always start with *get* or *set*. We need more intelligence if we plan to extract all of these configuration options from the code.

Finally, if we are concerned with knowing the current state of the configuration space at some point in time, we need a technique that captures an **accurate configuration snapshot** at runtime. Indeed it may not be straightforward to get this information from the system. In some of the applications we have studied (Firefox and LibreOffice), when the user modifies a preference value dynamically through the option menu, the change is reflected immediately in the dynamic memory and preference files. However, in our industrial application, the change made by the user will be stored temporarily shutdown and the new preference will take place on the next startup. Therefore the running configuration and the one reflected in the persistent memory after the application closes may be inconsistent.

2.1.2 Natural Language Processing and Information Retrieval

In Chapter 4, we present PrefFinder, a natural language based querying framework to identify configurable options. This sections defines the natural language processing and information retrieval terminologies we used throughout the thesis.

Soft words are individual dictionary words, such as *browser*, and *office*. *Hard words* are a super set of soft words. A hard word may contain a single soft word, such as *browser*. Other hard words consist of multiple soft words, which are joined together in the same case (e.g., *openoffice* and *codegen*), or by camel case (e.g, *targetPlatform* and *RecoveryList*). *Stop words* are words that do not provide domain relevant information in the context [3,16]. Words such as *doesnt*, *me*, *when*, *any*, *more* are some examples of stop words.

The classic information retrieval weighting scheme *term frequency-inverse document frequency* (*tf-idf*) [3,29,48] is often used to compute the similarity for a (query, document) pair. The scheme measures the importance of a word to a document. The following terminologies are used in the discussion. A user *query* (*q*) contains a bag of words and each word in *q* is a *term* (*t*). Each preference name can be thought of as a small *document* (*d*) that also contains a bag of words. A preference system consists of N preferences forms a *collection* (*c*) of size *N*. *Term frequency* ($tf_{t,d}$) is defined as the number of occurrence of a term *t* in the document *d*. The value of $tf_{t,d}$ equals to zero if *t* is not in *d*. *Document frequency* (df_t) is defined as the number of documents in the collection that contains the term *t*. The value of df_t equals to zero if *t* does not exist in any of the documents in the collection. On the contrary, *inverse document frequency* (idf_t) is defined by the equation:

$$idf_t = \log \frac{N}{df_t},$$

where df_t is the document frequency of term *t* and *N* is the number of documents in the collection. Note that if a term exists in many documents, it often carries less discriminating

power (df_t is large, and thus makes idf_t small). Hence, idf_t can be use the effect of terms that appear in too many documents.

The *tf-idf* weight for a term in *d* is defined by the equation:

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t,$$

which is the product of the term frequency and the inverse document frequency for that item (weight equals to zero if the item only occurs in *d* but not *q*). As can be seen, a term in *d* would have a heavier weight if it occurs many times in a few documents (both $tf_{t,d}$ and idf_t are large). The similarity score for a (query, document) pair is computed as the sum of *tf-idf* weights for all the items that occur in both the query *q* and the document *d* by the following equation:

$$\text{score}(q, d) = \sum_{t \in q} tf\text{-}idf_{t,d}$$

In our version of the algorithm, we impose an additional scale factor on top of the *tf-idf* weighting. See Chapter 4 for the details.

2.2 Related Work

We provide an overview of several areas of research that are closely related to this work. The role of software users and essential information in bug-fixing has been emphasized in several studies [2, 4, 47, 59]. Bettenburg et al. [2] found that there is usually a strong mismatch in bug reports between what developers need to reproduce and fix a bug and what is provided by users. Herbold et al. [20] developed a tool to capture usage logs for replaying bugs. Other work tries to reproduce field failures [5, 24], however the focus is on using the call graph. None of this work tries to capture the software configuration used during the failure.

Several researchers have been focusing on extracting configuration options from code. Rabkin et al. [45, 46] propose a method to statically detect system configurations, but as already mentioned this analysis works on a single language (Java) and assumes that all configurations are contained in a single class. Yin et al. [56] conducted empirical studies to understand the configuration errors in commercial and open source systems. Zhang et al. [57] have proposed a technique to diagnose crashing and non-crashing errors related to software misconfigurations. Again their tool only works on a single language (Java) and the configurations they study are simple. We look at more complex configuration spaces with multiple languages and multiple preference layers, etc.

From a traceability perspective, there has been a large body of research [6, 10, 19, 26, 28, 30], but most focuses on the traceability of requirements, architecture and quality attributes. Recent research has looked at extracting traceability for feature models (a type of configuration model space) [11, 25], but this has been achieved only through documentation, rather than by examining the multiple layers of the software preference space. We believe some of this work can be leveraged for configurability.

There has been considerable work on using natural language to improve code documentation and understanding [15,16,21,22,49] and to create code traceability links [12,27,38]. In addition, recent work on finding relevant code, uses search to find code snippets that satisfy a given purpose [31, 52]. While this work is related to our problem, the techniques assume that there is a large code base to explore and leverage this in their similarity techniques; we want to associate behavior with identifier names with little or no context. In the long run, we believe that being able to identify desired preferences can enhance traceability (e.g. between menus items and code elements), but before this is possible, we need to first be able to extract these preferences individually.

Finally, there has been a large body of work in the software testing community that demonstrates the need for configuration-aware testing techniques [39, 41, 42, 55] and pro-

poses methods to sample and prioritize the configuration space [7, 14, 51, 55]. There has also been recent work that uses configurability as a way to avoid failures through self-adaptation [17]. But all of this work assumes that the configuration model is known (or is somehow extracted).

Chapter 3

An Analysis of Configurability in Real World Systems

In this chapter we present a case study that analyzes the true configuration space of highly configurable software systems. Some of the work presented in this chapter has been published in [23].

3.1 Motivation

As we work more and more with highly-configurable systems in practice, we have discovered common issues that arise which make available configuration-aware techniques insufficient. For instance, there usually is no single document that describes the complete set of possible configuration options. We can examine external preference files, but we find that there may be multiple files, and they still tell only a partial story because there are *hidden* (but valid) preferences found only in the source code. We can try to use an analysis technique such as those proposed by Rabkin et al. [45, 46] to reverse engineer a complete mapping of our configuration space, but many applications are written in multi-

ple languages (e.g. C++, Java, and JavaScript) and often use aliasing to refer to preference names, neither of which are supported by existing techniques. Finally, if we assume that we can somehow obtain the ground truth model of the configuration space, then in order to manipulate the configurations for testing and debugging, we need mechanisms to automate this process, as well as ways to capture *which* configuration was active during a failure. Again, we have learned that the complexity of real software makes this difficult – configurations can be modified and viewed from multiple locations, and are found in both dynamic and static structures. Finally, we have discovered that it is possible for the static structures to be out of synchronization with the dynamic ones at the time of failure.

Faced with the complexity that we have described informally so far, we want to quantify how often we see these problems with the aim of developing a generic model of how modern highly-configurable software is structured and manipulated. We also want to know if there is a ground truth for the configuration model and dynamic configuration states in modern configurable systems. We present an analysis in this chapter that we have developed for this purpose.

3.2 Case Study

We present a case study to help us extract a general model of modern, highly-configurable systems. Our study has two main objectives. First, we want to quantify the complexity of the configuration space and what mechanisms are used to define and manipulate this space. Second, we want to understand what are the challenges that we will face as we develop configuration-aware testing and debugging techniques. To address these issues we will center our study around answering the following research questions.

RQ1: *What is the complexity of the configuration space in modern configurable software systems?*

RQ2: *How are configuration options structured, changed and accessed by the user in these systems?*

RQ3: *Are the selected configuration options synchronized between the different parts of the system and throughout the lifecycle of program execution?*

3.2.1 Software Subjects Studied

We have selected three different software systems to study. The first subject, Firefox, is an open source web browser which works on multiple operating systems and has over 300 Million users worldwide [36] and over 9.6 Million lines of code [37]. The second subject is LibreOffice. It is an open source office productivity suite consisting of a word processor, spreadsheet application, presentation tool, drawing application, math formula tool and database [13]. LibreOffice has 6.8 Million lines [37] of code and 25 Million users worldwide estimated by The Document Foundation in 2011 [53]. The third subject is a large real-time embedded software system developed at ABB (called ABB_c hereafter). ABB_c has approximately 10 Million lines of code, is highly-configurable, and has more than 58 modules; each module defines a subsystem that implements a different set of functionality of the system.

3.2.2 Study Design

To answer our research questions, we collect configuration information from both a static and dynamic perspective on each system. We manually study all artifacts that are publicly available to users, including documents (e.g., user manuals and online help pages), software option menus on the user interface, preference files and source code. We also utilize tools or APIs that have been provided to manipulate internal data structures that hold configuration information. For ABB_c we have a user manual that is written for those

who will modify and change preference files. In addition, we have asked questions of developers to confirm our assumptions. In Firefox we utilize the source code, examine the internal dynamic data structures via an API call when the application is running, as well as study the `about:config` page (a utility for modifying configurations). We also study the `Options` menu, the SQLite database that holds page specific preferences, and online documentation. For LibreOffice, with the help of online documentation, we study the preference files and used an API to connect to the dynamic data structures when the program is running. To answer RQ1, we calculate the ABB_c configuration space based on the user manual and we calculate the configuration space for Firefox and LibreOffice by querying the dynamic data structures at runtime.

When we collect the configuration information, we make some assumptions. First, constraints between options are ignored. We realize that this might over approximate the configuration space, but extracting the exact configurations options may not be feasible without in-depth knowledge of each system. Second, the plug-ins (add-ons) are not included in our calculations. In Firefox and LibreOffice, we build clean versions of the system from source code for study. Any default plug-ins that come with those will have their configuration options included, however no additional plug-ins are enabled. To calculate the number of values associated with an option, we have detailed information for many of the configuration options in the ABB_c manual. However, when they are not available, and for Firefox and LibreOffice, we use a set of rules to come up with a small set of categories. For Boolean configuration options we use *True* or *False*. For integers we use a 'default value', a 'non-default legal value' and an 'illegal value', resulting in 3 values. For strings we use 'no string', an 'empty string' and a 'legal string', again resulting in 3 values. In ABB_c we have some strings with constraints. For these we use 4 values by adding an 'illegal string'. This partitioning may underestimate the true configuration space, (it is a conservative model), but it is consistent with prior work [7].

For RQ2 and RQ3 we analyze the systems further and experiment with the various ways that one can modify configurations when the system is not running. We also analyze what happens if configurations are modified while it is running as well as what occurs with the changed configuration options during startup and shutdown. We examine some of the preference setter code and also look for hidden preferences that may not have been exposed earlier. We look at both menu access as well as file access. We also use the specialized tools such as the `about:config` to interface with Firefox and the ABB tools (denoted as ABB_a and ABB_b) to interface with ABB_c.

3.2.3 Threats to Validity

As with any study there are threats to validity which we document here. First, we have only studied three software systems. While we believe they are different enough (one is an industry application while two are open source applications with different sets of developers) we can not be sure that our results will generalize to all configurable applications. Our second main threat is that we are not developers of these systems so we have relied on the documentation and code to extract the information that we need. With ABB_c we were able to confirm our questions with developers. In the Firefox and LibreOffice environment we do not have this as a source of validation. But we used third party APIs that are commonly used to interact with the configuration environments and made an effort to validate our result internally. We have made the tools we used to query Firefox and LibreOffice available online as well as the artifacts that we have obtained to reduce this threat. Finally, we could have measured different elements for this study, but feel that the set of metrics we collected supports our research questions.

Table 3.1: Quantifying number of preference files and preferences of ABB_c, Firefox and LibreOffice

	ABB_c	**Firefox**	**LibreOffice**
Operating System	Embedded System	Ubuntu 12.04	Ubuntu 12.04
Version	-	Mozilla Firefox 27.0a1	LibreOffice 4.0
LOC (M)	10.0	9.6	6.8
Primary Languages	C++(3.7%), C(29.6%),C#(8%)	C++(41%),C(21%), JavaScript(16%),Java(3.1%), Python(2.7%), Assembly(1.2%), Shell script(1%)	C++(82%), Java(6%)
Total Pref Files	6	11	193
Total Prefs	524	1957	36322

Table 3.2: Categorization of configuration space for ABB_c and Firefox. The total number of preferences are shown as c^n where c is the cardinality of the preference (number of values) and n is the number of times we have this cardinality). We have combined like cardinalities together therefore the total *boolean* values for example may include some from the *others* category

Types	ABB_c	**Firefox**
Boolean (2)	92	846
Integer (3)	271	517
String (3)	27	594
String with condition (4)	110	–
Others	24	–
Total	$2^{96}3^{303}4^{114}6^{4}7^{1}8^{3}9^{1}16^{1}18^{1}$	$2^{846}3^{1111}$

Table 3.3: Categorization of the configuration space for LibreOffice broken down by module

Types	**Writer**	**Calc**	**Impress**	**Draw**	**Math**	**Database**	**Others**	**Total**
Boolean (2)	201	58	69	44	77	44	3940	4433
Integer (3)	157	43	26	22	110	15	5087	5460
Others	298	70	32	3	141	167	25718	26429
Total	$2^{201}3^{455}$	$2^{58}3^{113}$	$2^{69}3^{58}$	$2^{44}3^{25}$	$2^{77}3^{251}$	$2^{44}3^{182}$	$2^{3940}3^{30805}$	$2^{4433}3^{31889}$

3.3 Study Results

We now present our results for each of the three research questions. Supplemental data for the open source applications can be found on the associated website (`http://cse.unl.edu/~myra/artifacts/Configurations-2014/`) [23].

3.3.1 RQ1 Configuration Complexity

To answer RQ1, we turn to Tables 3.1, 3.2 and 3.3. Table 3.1 provides the basic statistics for our applications. It first shows the operating system and versions of the two open source applications. We then list the primary languages that are used in each application. We show all languages that make up at least 1% of the code. We leave out markup languages such as XML or XUL. All three applications consist of at least two languages. Firefox has the most with C++, C, JavaScript, Python, Assembly and some shell script. LibreOffice has both C++ and Java. ABB_c has a mixture of three languages, C++, C and C#. We also list the number of preference files that are used to store the current set of preferences and that are read at startup. As we see, this ranges from 6 files in ABB_c to 193 in LibreOffice (there are six preference files in ABB_c, but we were unable to access one of them, so all of the computation that follows uses only five files). Finally, we list the total numbers of unique preferences that we counted in each of these applications. This ranges from 524 in ABB_c to 36,322 in LibreOffice.

We next look at Tables 3.2 and 3.3. We show a breakdown of the configuration options by the data types and number of values associated with each type. Table 3.2 has data for ABB_c and Firefox. As we can see, we have only three types in Firefox resulting in 846 boolean options and 1,111 options of either integer or string, each with three values. The total configuration space is equal to $2^{846} \times 3^{1111}$. ABB_c has a variety of cardinalities for its configuration options. We have a more exact model due to better documentation. Our total configuration space for this application is 6.46×10^{259}.

Finally we look at Table 3.3 which shows the configuration options in LibreOffice broken down by individual modules within the suite of tools. This is based on the hierarchical path used to display the configuration option name. For instance all of the preferences under `Writer` have the prefix `org.openoffice.Office.Writer`. We do not believe

that all 36,322 would be used together in any test or debug model. Instead one would test an application such as `Writer` individually. Although we can identify which preferences belong to specific applications such as `Writer` or `Calc`, there are some categories such as `UI` which may be shared among applications. These all fall into the `Others` category. The complete categorizations are contained on our website.

3.3.1.1 Additional Complexity for ABB_c

ABB_c has preference files that contain additional information not found in the open source applications. This is because it is an embedded system with configuration options that can be customized for different drivers or ports. The number of devices and ports is open ended. The two additional pieces of information in these preference files are *category* and *instance*. Certain preferences are grouped into a category, and for each category we have one or more instances that consist of the same set of preferences. Each category may contain multiple instances, therefore one preference can appear multiple times. To understand this better, we can consider a situation where each instance is associated with a specific hardware or virtual device. Some devices are in the same category, thus have the same set of preferences, however the device that is being controlled differs.

An example of a snippet of the ABB_c preference file is illustrated in Figure 3.1 (the names are changed for proprietary purposes). There are five options in this figure (bold fonts): **Name** (string), **Count** (integer) , **Unit** (string) , **Length** (integer), and **Status** (boolean). **Name** and **Count** are grouped under `CATEGORY A`, while **Unit**, **Length**, and **Status** are grouped under `CATEGORY B`. There are three instances in `CATEGORY A`: in the first instance (line 3), the **Name** is assigned with value *x* and **Count** is assigned with *2*; in the second instance (line 4), the **Name** is assigned with *y* and **Count** is assigned with *5*; in the third instance (line 5), the **Name** is *z* and **Count** is the default value. [1] Similarly,

[1]The ABB_c user manual states that "if the option is assigned the default value, then it will not be listed

there is one instance in `CATEGORY B` (line 8): the option **Unit** is assigned with *X*, the **Length** is assigned with *10*, and the **Status** is assigned with *ON*.

Table 3.4 shows the number of configuration options grouped by *categories* and the number of categories for each preference file. In this thesis, when we compute the configuration space shown in Table 3.2, we made a conservative assumption that all options will appear a single time (regardless of instances), to make it in consistant with other systems.

```
1.    #
2.  CATEGORY A:
3.   -Name "x" -Count "2"
4.   -Name "y" -Count " 5"
5.   -Name "z"
6.   #
7. CATEGORY B:
8.  -Unit" X" -Length "10" -status "ON"
```

Figure 3.1: Example of ABB_c Preference File

Table 3.4: Number of options grouped by categories in ABB_c

Preference Files	Number of Categories	Number of Options
File 1	3	26
File 2	11	50
File 3	10	78
File 4	7	22
File 5	39	348
Total	70	524

in the configuration file" and this is why the third instance only has one option explicitly written.

3.3.2 RQ2 Configuration Access

We begin answering RQ2 by examining the structure of one of our open source systems, Firefox. Figure 3.2 shows this schematically. In this figure there are a number of preference files (both user and default) that contain values for specific preferences. During the application startup, the default configuration options are read (there are 1932 of them), and after that, the user preferences are read (there are 50 of them initially). These are read by the preference modules. The user can modify these on disk directly if they understand the format. The next time the application opens, these files will be read (assuming that they have not been overwritten in the meantime – see RQ3 for a discussion of that mechanism) and the preferences will be activated. The user can also open Firefox and use the `about:config` webpage to control (or look at) the preferences. If a user modifies a preference in the `about:config` it will be written to the user preference file and be set via the preference modules in the code. Additionally the user can go through the options menu. This contains only a subset of the full set of possible options, only 126 out of the 1957 (calculated in Table 3.1). We do not quantify (or discuss) the Add-on configuration options in this thesis, but these are also manipulated through a menu. Finally, there is an SQLite database which contains page-specific option settings for the browser (e.g. if a user zooms in on a particular website, this information will be stored for the next time they open that site).

The preference modules are accessible through a set of preference APIs. The APIs are used to interface with a dynamic hash table which contains all active configurations when an application is running. There is a 1 to 1 mapping of the preference files to the hash table, but an N to 1 mapping of the menu items. These are used as variables in the code and several names may map to the same individual option in memory. Finally the code itself (program modules) contain the ground truth for the configuration space. We have

discovered several options in the code that are *hidden*. These are options without default values that can be set if a user knows about them, but which do not appear in our results for RQ1 since they are not in the hash table or preference files unless explicitly set by the user.

We have analyzed the UI source code of the Firefox option menu and retrieved 126 preferences that are bound to the option menu UI elements. Listing 3.3 shows an example of binding the preference browser.startup.page, which specifies the start-up page when one opens Firefox, to a drop-down menu list in the option menu. As can be seen, only 6.4% of the total preferences exist in the option menu in Firefox.

We note that both the ABB_c and LibreOffice systems have similar structures, therefore, we do not show them all here, but an extraction of the general structure is illustrated in Figure 2.2 and introduced in Chapter 2.

We next investigate how configuration values are read in the code. First, we take a look at the APIs used to access the configurations in the code. In Firefox, the return value is almost always passed by reference. For example, the signature of a boolean preference access functions from the source file $prefapi.h$ under $/modules/libpref/src$ is shown in Listing 3.1. As we can see, the configuration option value $return\_val$ is passed as a pointer in the formal parameter list. The function returning value (i.e., $nresult$) is just an binary indicator of whether the actions defined in this function succeed or fail. This prevents us from using the techniques developed by Rabkin et al. [45, 46] because the preference type cannot be inferred by tracking return value types.

```
nsresult PREF_GetBoolPref(const char *pref, bool * return_val , bool get_default );
```

Listing 3.1: Return value is passed by reference

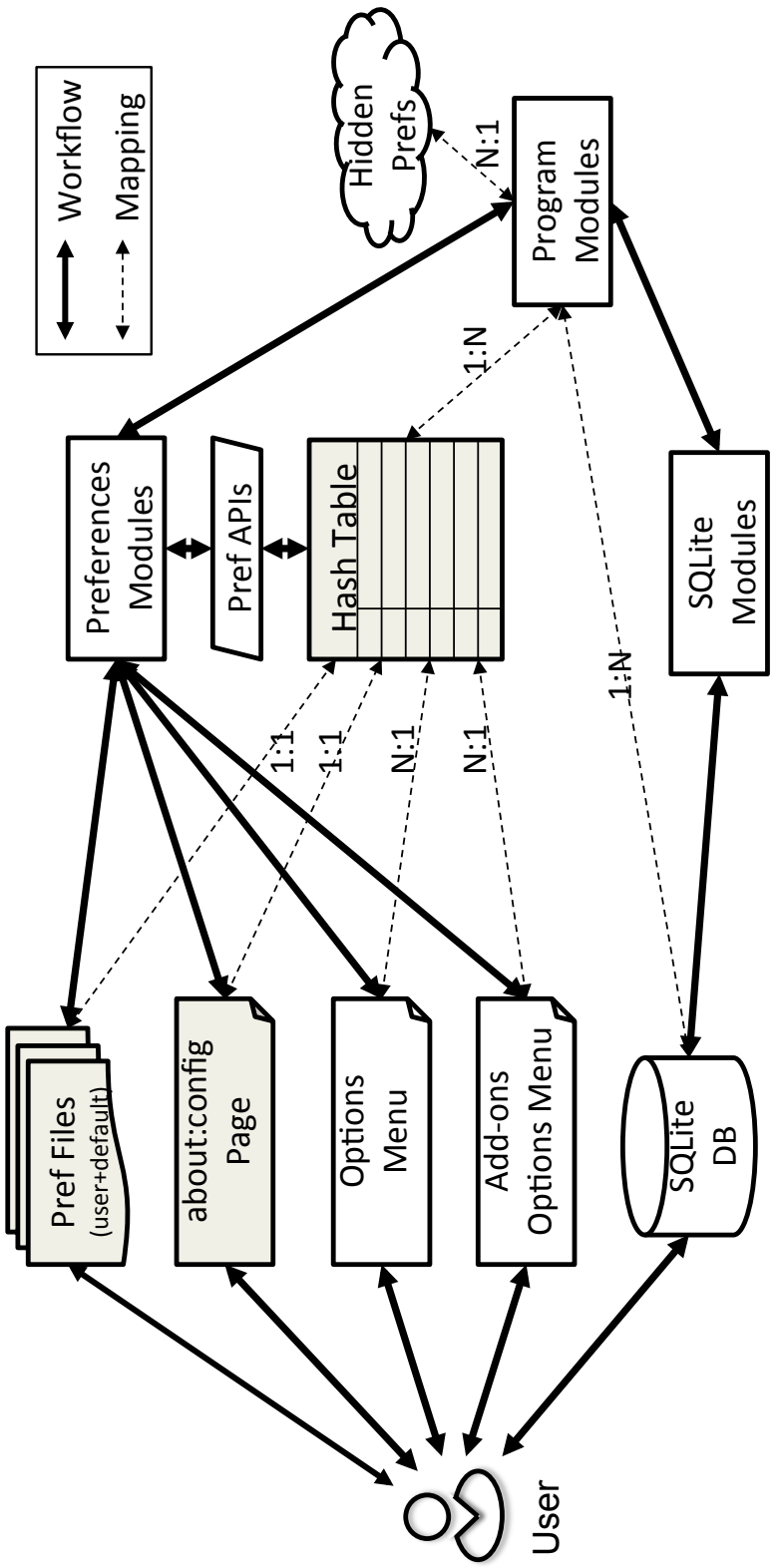

Figure 3.2: Firefox Configuration Structural Diagram

```
// nsBrowserContentHandler.js
var choice = prefb.getIntPref("browser.startup.page");
// nsBrowserGlue.js
Services.prefs.setIntPref("browser.startup.page", 3);
```

Listing 3.2: Query and update Firefox preferences using JavasScript

```
// main.xul
<preference id="browser.startup.page" name="browser.startup.page" type="int"/>
...
<menulist id="browserStartupPage" preference="browser.startup.page">
  <menupopup>
    <menuitem label="&startupHomePage.label;" value="1" id="browserStartupHomePage"/>
    <menuitem label="&startupBlankPage.label;" value="0" id="browserStartupBlank"/>
    <menuitem label="&startupLastSession.label;" value="3"
        id="browserStartupLastSession"/>
  </menupopup>
</menulist>

//main.js
let startupPref = document.getElementById("browser.startup.page");
...
startupPref.updateElements();
```

Listing 3.3: Query and update Firefox preferences using XUL

```
// String
rv = mPrefBranch->GetBoolPref("autoadmin.append_emailaddr", &appendMail);
// Variable
prefBranch->GetIntPref(kCookiesLifetimeBehavior, &lifetimeBehavior);
// Object macro
rv = branch->GetIntPref(DISK_CACHE_CAPACITY_PREF, &capacity);
// Function macro
rv = prefs->GetIntPref(HTTP_PREF("connection-retry-timeout"), &val);
// Class member
rv = prefBranch->GetBoolPref(externalProtocolPref.get(), &externalProtocol);
```

Listing 3.4: Different types of API preference name parameters

Second, the preferences are accessed via multiple programming languages. The Listing 3.2 and 3.3 show two examples of the Firefox source code interfacing with the preference system via JavaScript and XUL respectively. The JavaScript performs most of the manipulation, but the XUL code interfaces and dereferences the preference name.

Third, the preference name can be in various forms when passing to preference APIs. The name of the preference is usually passed as the first parameter to the preference APIs. Listing 3.4 shows a few examples of passing the preference name as a string, a variable, an object macro, a function macro, or a class member.

Finally, we show examples of hidden preferences. In the `String` example in Listing 3.4, the preference autoadmin.append_emailaddr appears in the source code, but it does not exist in any preference files unless added by the user. We consider it as a *hidden preference*. Preferences shown in Listing 3.5 are some other examples of hidden preferences from Firefox source code. Our configuration space analysis (RQ1) misses these preferences. We do not know how many exist in Firefox.

```
pref.browser.homepage.disable_button.bookmark_page
pref.browser.homepage.disable_button. current_page
pref.browser.homepage.disable_button.  restore_default
```

Listing 3.5: Hidden preferences

We also investigate how configuration values are read in code in ABB_c. First, there is a configuration manager class (written in C) that reads the values at different levels: it may read values of a single preference, it may read a single instance that contains a couple of preferences, or it may read all instances that under the same configuration category. Just like in Firefox, all these values are passed by reference. Second, the name of the preferences can be in various forms, such as string, variable, and macro. Finally, there are several configuration options that are accessed in the code but not in the document (hidden preferences) and there are also some configuration options that are in the document but are never read in the code (dead preferences).

Table 3.5: Number of configurations accessible at different layers

System		**Static View**		**External Control**	
	Table 3.1	Code	Manual	Pref. Files	Menu
ABB_c	**524**	$428 + 166$	**524**	< 524	< 524
Firefox	**1957**	> 1957	NA	> 1957	126

Table 3.5 summarizes the number of configuration options that are accessible at different layers (defined in Figure 2.2). The first column (Table 3.1) shows the values we obtained for RQ1. The last column (Menu), is used to represent configuration control via menu in Firefox and via ABB_a and ABB_b in ABB_c. For ABB_c there are $(428 + 166 = 594)$ options accessed in code. 428 options are also described in the manual, but 166 options only appear in code (hidden preferences), and $(524 - 428 = 96)$ options only appear in the document (dead preferences). This shows that the document is not updated accordingly as the code is changed, although the document is a very important artifact that tightly connects the system with customers. We do not have accurate numbers of the preferences accessible

by external control elements, but quote the manual which says “if the option is assigned the default value, then it will not be listed in the preference file.”; there are also preferences not in ABB_a or ABB_b given that “some configurations have to be changed in preference files”.

3.3.3 RQ3 Configuration Synchronization

To answer RQ3, we map the lifecycle of a running application to understand when and where its configurations are synchronized between its layers. We model three distinct phases, *startup*, *runtime*, and *shutdown*. Figure 3.3 shows the behaviors of Firefox and LibreOffice, and Figure 3.4 shows the behaviors of ABB_c. The numbers on the leftmost side specifies the number of preference files in different groups of files. Solid arrows represent direct connections, while dashed arrows indicate the need for a mapping/traceability.

In all three systems at startup, the configurations are read from persistent storage (configuration files) and loaded into memory. There is a specific order in which these are loaded. If the same configuration options are repeated, set to different values, the last one read will be the one which holds. While the applications are running, a user can modify the configuration files directly. This is not immediately reflected in the dynamic memory. If a failure occurs at this point the persistent memory is out of sync with the dynamic. In all three systems the user can also dynamically modify the configurations while the application is running. In Firefox and LibreOffice these will take effect immediately and be written back to the preference files. In ABB_c the dynamic memory is not updated. The changed configurations are held in temporary memory and take effect at the next startup.

On shutdown, in Firefox and LibreOffice the dynamic memory overwrites the current preference files before the application closes. In Firefox the user preference file is overwritten, but the default ones are not. This means that if a user modified the user preference files

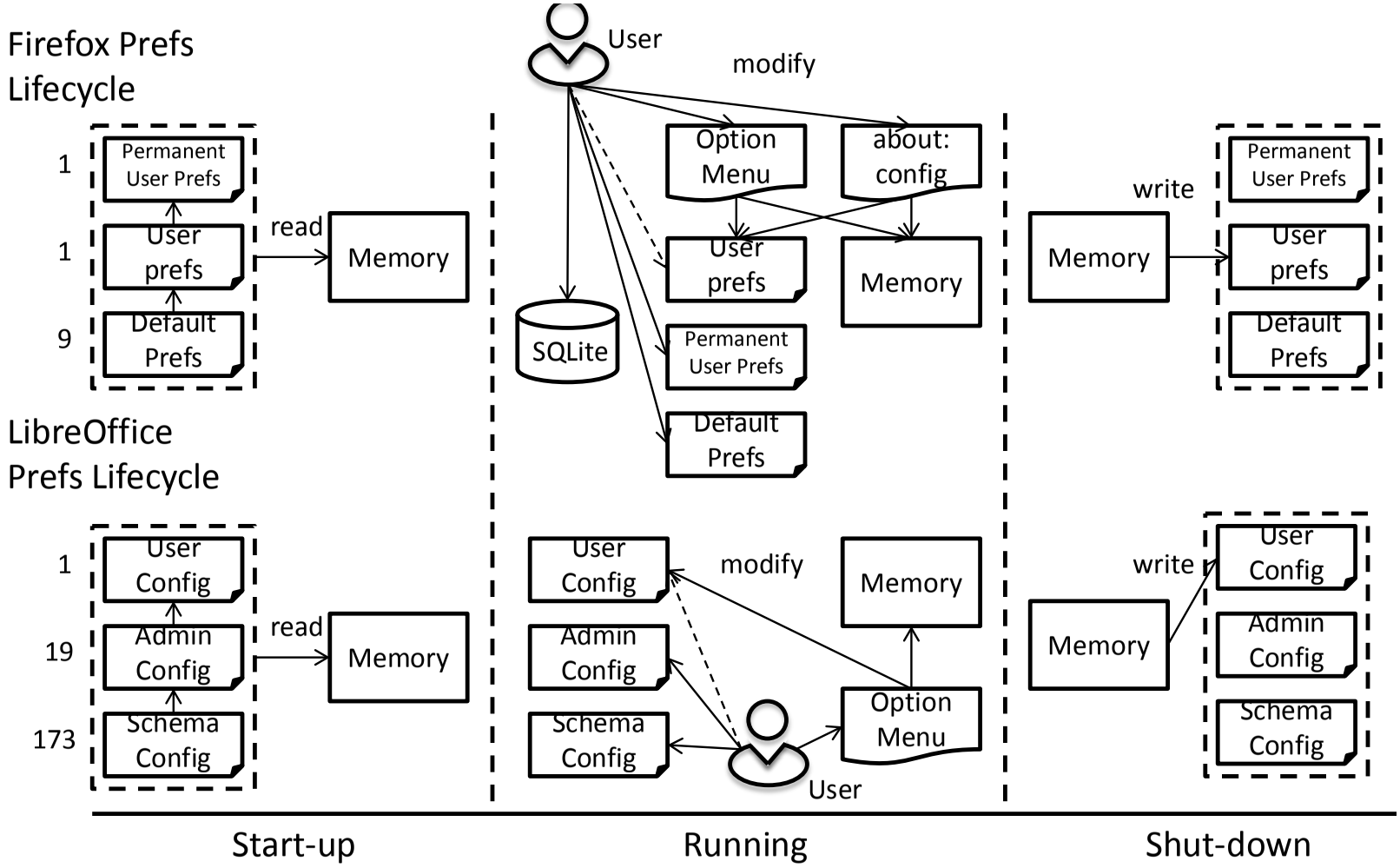

Figure 3.3: Firefox and LibreOffice Lifecycle Diagram

during runtime, those changes will never be seen (not even on the next startup). However, if they modified other preference files they will appear on next startup.

ABB_c has a more complicated "restart" behavior, described as follows. When the system is restarted normally (denoted as start-I): the current system will be stopped. All system preferences will be saved. Restarting this way will activate any configuration changes. A second option is to restart and select another configuration (start-II). In this case the current system will be stopped. All system preferences will be saved, so that the system state can be resumed later. The last restart is to restart and return to default settings (start-III). After restart, the system state will be resumed but any changes done to system preferences will be lost. Instead, system preferences are read from the originally installed system on delivery.

Furthermore in ABB_c there are three sets of preferences: active (loaded by default), backup, and default. During startup, instead of loading different sets of preferences in order (as happens in Firefox and LibreOffice), the system only loads one set of preferences into memory, based on the type of restart. During normal start and start-I, the active preferences are loaded, during start-II, a selected set of previous backup preferences are loaded, and during start-III, the factory default preferences are loaded. During run time, the users can make configuration changes in preference files directly, or through ABB_a or ABB_b, but changes will not take effect until a restart. The changes will be stored temporarily in a memory different from the active preferences. Users can also save the currently active preferences as a backup. Finally, all changes made at the runtime will be written back into the active preference files when the system is normally shutdown or restarted in I or II. Note that if the users select a start-III, all changes will be lost.

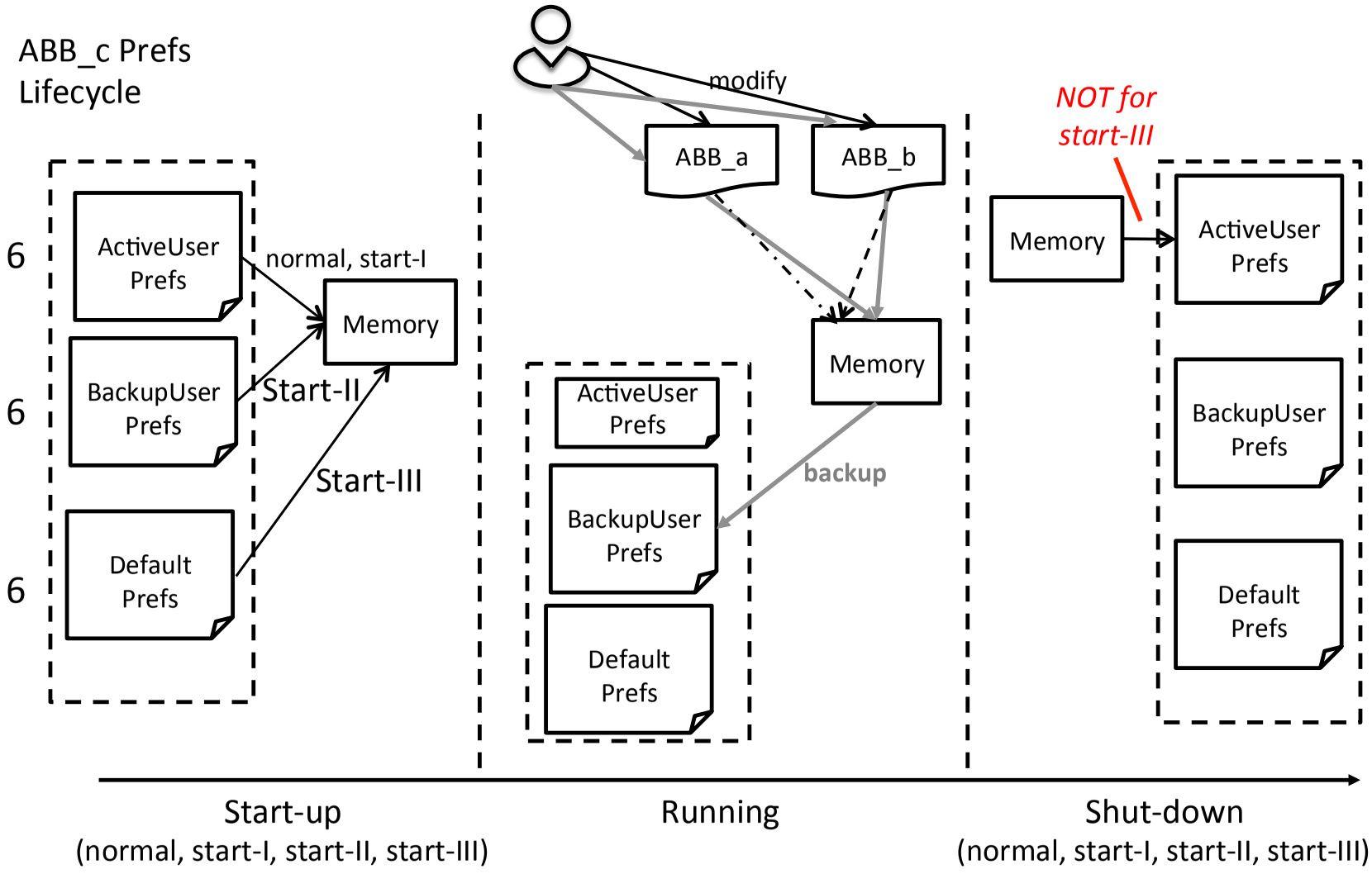

Figure 3.4: ABB_c Lifecycle Diagram

3.4 Discussion

In this section we summarize the implications and lessons learned from our study. The first two lessons learned are geared towards practitioners since they reflect the state-of-the-art. The last two provide a roadmap for researchers who plan to develop new tools and techniques for configuration-aware testing and debugging.

1. Configuration Modeling Should Merge Multiple Layers

We return to our first question of how one can model the full configuration space when performing testing and debugging. Although the application code is the *ground truth*, the maintenance engineers may not always have access to code. If instead we use the user manuals/documentation, we most certainly miss out on some configurations. Moreover, in the applications studied, the menu on the user interface contained only a small subset of the configuration options. While these might contain the most widely used preferences, they do not provide a true indication of the real configurability of a system. Finally, we can use the persistent configuration preference files, but we must first understand how (in what order) and when these are activated in the dynamic system. Two issues that have arisen during our analysis are those of *hidden preferences* and *dead preferences*. These constitute a small part of the configuration space model, but one should be aware of their potential existence. Given the results of our study, we believe that to obtain an accurate model of the configuration space one should consider and merge multiple artifacts which includes preference files, menus and documentation. Additionally, since documentation is the primary artifact a user would read, it should be updated as the design and code changes in a timely manner, particularly when it comes to system testing or other configuration related tasks.

2. Configuration Traceability is a Necessity

Given the variety of places that configurations are accessed and mentioned, it seems

that the task of simply setting a configuration option requires deep knowledge of the application. If we return to our example, Firefox, one needs to know the mapping of menu names to preference variable names to modify them automatically. Furthermore we have seen (both in Firefox and ABB_c), a many-to-one mapping of variables in the code and preference files and dynamic memory. Providing traceability mapping between elements of the configuration manipulation mechanism are essential to making configuration-aware techniques work.

3. Analysis Tools Need to Cross the Programming Language Barrier

As we have seen, the current state of research in analysis for extracting configurations from code expects a single programming language and single class files where the configuration information code (such as setting and getting configuration) lies. Yet this is not realistic for the large scale subjects that we have studied. Our configuration options are manipulated and referenced across programming language barriers and in multiple modules. We need, therefore, new analysis techniques that cross these boundaries, can handle aliasing, and that use additional heuristics to identify the actual getter and setter code.

4. Configuration State Capture or Approximation Techniques are Needed

As we argued above, we need a way to capture the active configuration when the system fails so that we can reproduce and debug the failing test case. Each of the three systems we studied, allows the user or maintenance engineer to modify the configurations both externally or internally during runtime. While our open source applications update the memory and files immediately, in our industrial application, the configuration is not activated until possibly startup (with the exact behavior dependent on the type of reboot selected). Even if we understand how the configuration manipulation works, there is the possibility of race conditions in all of the applications, depending on the exact timing of the configuration modification and failure. It is also possible to make changes to external files for modifica-

tions at startup, yet these may be overwritten during a normal shutdown. In order to extract the ground truth of the *configuration at failure*, monitors are needed that capture this information. But these may incur overhead and cause concerns for privacy. Alternatively, we know that the persistent memory contains a large portion of the correct configuration space, so algorithms that work from this point and search close by may be useful for reproducibility. Research has shown that failures tend to have *feature locality* [17], so it is possible we can leverage some of those ideas for this work.

3.5 Summary

In this chapter we analyzed a highly-configurable industrial application and two open source applications in order to quantify the true challenges that configurability creates for software testing and debugging. We find that (1) all three applications are multi-lingual, hence static analyses need to cross programming language barriers to work, (2) there are multiple access points and methods to modify configurations, implying that practitioners need configuration traceability and should gather and merge metadata from more than one source and (3) the configuration state of an application on failure cannot be reliably determined by reading persistent data; a runtime memory dump or other heuristics must be used for accurate debugging. We also provided a roadmap and lessons learned that will help practitioners better handle configurability now, and that may lead to new configuration-aware testing and debugging techniques in the future.

Chapter 4

PrefFinder

In this chapter we present the recommendation aspect of this work. We introduce PrefFinder, a natural language recommendation framework.

4.1 Overview

Figure 4.1 shows an overview of the PrefFinder framework. The application view is responsible for interacting with system preferences and interfacing with the user. Preferences can be extracted from sets of configuration files, through a static analysis [44], or by hooking into a dynamic data structure such as the Firefox hash table. It is also possible, that an external resource such as an online help system could be used, so that there is an explanation for each preference as well (such a help utility does exist online for `about:config` and is something we intend to include as future work). PrefFinder takes the system preferences and first parses these into sets of keywords. It also accepts user queries in the form of natural language. A series of back-end databases can be used to increase the effectiveness of PrefFinder. For instance, it can include different types of dictionaries as well as a lexical database to allow for synonyms or other "close" matches. The parsing and ranking algo-

rithms extract the meanings from the queries, search the parsed preferences and return a ranked list of suggestions. The rest of this chapter explains the framework in more detail.

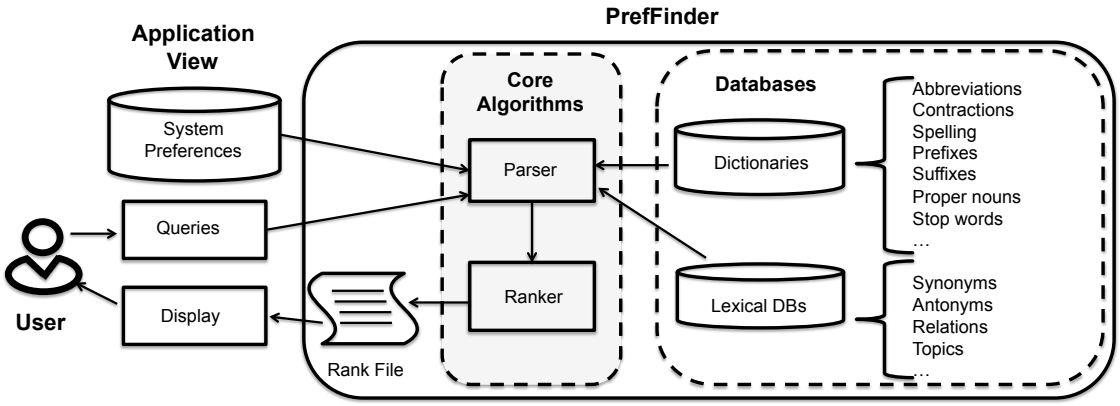

Figure 4.1: PrefFinder framework architecture

4.1.1 Application View

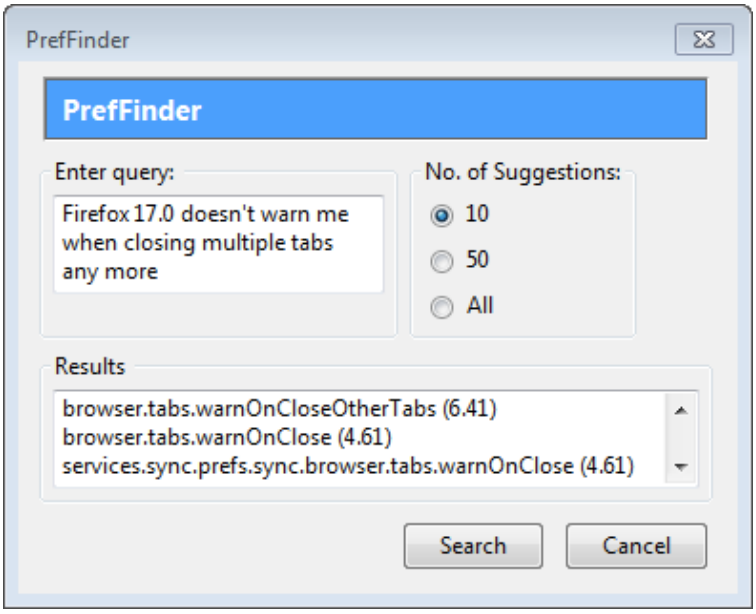

Figure 4.2: PrefFinder prototype user interface

The front-end of PrefFinder interfaces with both the user and the target software sys-

tem. This can be a command line application to allow automation for multiple queries at a time, or it can be an interactive application. Figure 4.2 shows our prototype extension for Firefox as it appears in the Windows operating system. The user will enter a short description in English about what features or functionality of the system they want to customize, and specify control parameters such as the number of results to display. In this example, the user is interested in seeing the first 10 results for the option that forces Firefox to warn someone when closing more than one tab at a time. The query "*Firefox 17.0 doesn't warn me when closing multiple tabs any more*" is a real question that someone asked on the Firefox Support Forum [33]. This behavior is controlled by the preference *browser.tabs.warnOnCloseOtherTabs*. Note that user may enter arbitrary English sentences with different punctuation, numbers, mixed-case letters, and using different forms of the language such as present participle (*closing*) and plural (*tabs*).

The results are returned in rank-order (with a value showing the score). As can be seen the first option has a higher rank (6.41) than the next two options.

4.1.2 Parser

Once the query has been submitted and the preferences read, there are two separate parsing activities that occur. The first one, only needs to be performed once (assuming that new preferences are not added during the running of PrefFinder). The second parsing occurs for each query. We discuss the preference parsing first and follow this with a discussion of the query parsing.

4.1.3 Preference Name Parsing

System preferences are often stored as name-value pairs [44]. For instance, the preferences in Firefox and Eclipse are stored in similar formats with the name of the prefer-

ence and its current value (e.g. *true*). LibreOffice uses a more complex XML format to store preferences, but the underlying format can still be seen as a name-value pair. Since we are not interested (at this time) in the values, we focus on parsing the names of the preferences. We adopt the commonly used information retrieval terminology in our following discussion [3, 16]. (Definitions of some terminologies used in this thesis can be found in Chapter 2). Preference names are usually represented as arbitrary strings, such as *browser.link.open_newwindow* in Firefox, *org.eclipse.jdt.core.compiler.codegen.targetPlatform* in Eclipse, and */org.openoffice.Office.Recovery/RecoveryList* in LibreOffice. Similar to program variable identifiers, a preference name must be a sequence of characters without any white space. In order to improve the readability, *soft words* within a preference name are separated by *word markers*, such as a period(.), underscore (_), dash (-), backslash (/), or are separated by the use of camel case letters. Using markers to split words is the first (trivial step). After splitting words at word markers, the remaining identifiers are called *hard words*.

To incorporate meaningful (code related) words to use during parsing in PrefFinder, we compiled a dictionary from the dictionary used in [21], which is derived from *iSpell* [18], and a list of computer science acronyms and abbreviations (such as *SYS* and *URL*) [1]. We also adopt a prefix list and a suffix list from the work of [15] to identify commonly used prefixes and suffixes (such as *uni-* and *-ibility*). Our dictionaries are available online (see Section 4.2).

In the related work on source code mining, several variants of splitting algorithms are used. In PrefFinder, we use a two step process for finding identifiers. After the initial separation by word markers, we first use a camel case splitting algorithm (Camel Case). We found that this often does not provide a clean split, so we have developed three additional *same case* splitting algorithms based on [15]. We evaluate these in our study. We use a forward greedy approach (Greedy) as was described in [16]. We also use a backward greedy

algorithm (Backward), which is a modification of this, and finally we tried a dynamic programming approach (DP).

4.1.3.1 Camel Case Splitting

We based our camel case algorithm on that from [15]. In that work they use the frequency of words to help rank which splits to make. Since we do not have this ability (i.e. we have no code to match), we consider all splits and then choose based on our reference dictionaries as described next. Our algorithm takes a hard word string s and the dictionary d as its inputs, and then outputs a space-delimited s. The algorithm loops through s from the beginning to the end sequentially to identify proper split positions. Note that s is kept intact if it does not contain any camel cases.

When the algorithm detects a pattern where a lowercase letter $s[i]$ occurs immediately before an uppercase letter $s[i+1]$, a space is inserted between these two letters. The algorithm then continues to process the rest of s. If the algorithm finds a pattern where a uppercase letter $s[i]$ occurs immediately before a lowercase letter $s[i+1]$, there are typically two scenarios. First, if there is just a single uppercase letter in front of $s[i+1]$, then no split is required. Thus, hard word *checkDefaultBrowser* would be split into *check Default Browser*. Second, if there is a sequence of uppercase letters before $s[i+1]$, we need to decide whether to split before or after $s[i]$. The algorithm first attempts to split before $s[i]$. The split is committed if either side exists in d. However, if this step fails to commit a split, then the algorithm attempts to split after $s[i]$ to see if any side exists in d. No split is made if both attempts fail. As a result, *HTMLDocument* and *XMLserializer* are split into *HTML Document* and *XML serializer*, respectively. Note that the algorithm favors the split before $s[i]$, since it is the more commonplace camel case practice.

4.1.3.2 Same Case Splitting

In the second step of parsing, each resulting hard word is further split using one of the same case identifier splitting algorithms. As was the camel case algorithm, these too are modifications from [15]. Again, our algorithm differs slightly since we do not have source code to mine. We also propose to split from the back end of the word, first and last we use an optimization approach (dynamic programming).

Algorithm 1 GreedySplit

```
1:  Input same-case string s, dictionary d
2:  Output space-delimited string s
3:
4:  if length(s) ≤ 1 ∨ s ∈ d then
5:      return s
6:  end if
7:  i ← 0, j ← 0
8:  while i < length(s) do
9:      if s[0 : i] ∈ d ∧ ¬isPrefix(s[0 : i]) then
10:         j ← i
11:     end if
12:     i ← i + 1
13: end while
14: if j = 0 then
15:     return s[0] + GreedySplit(s[1 : length(s) − 1], d)
16: else
17:     return s[0 : j] + " " + GreedySplit(s[j + 1 : length(s) − 1], d)
18: end if
```

Algorithm 1 shows the pseudocode of the forward greedy algorithm. It takes a same case identifier s and the dictionary d as inputs, and outputs the space-delimited s. If s is empty, a single letter, or a soft word, there is no need to split (line 4-6). Otherwise, it loops through s starting from the beginning and tries to find the longest prefix that happens to be a soft word (but cannot be any common prefix) (line 8-13). If such a prefix exists, the split is made and the algorithm is recursively called on the remaining substring (line 17). However, if such a prefix does not exist, the algorithm is recursively called on the

remaining substring that starts from the second position (line 15). As a result, Greedy is able to correctly split identifiers such as *browserid* into *browser id*. However, Greedy incorrectly splits *casesensitive* into *cases ens it iv e*, because it recognizes *cases* as the longest prefix soft word during the first iteration, and thus breaks the remaining substring apart.

Algorithm 2 BackwardSplit

```
 1: Input same-case string s, dictionary d
 2: Output space-delimited string s
 3:
 4: if length(s) ≤ 1 ∨ s ∈ d then
 5:     return s
 6: end if
 7: i ← length(s) − 2
 8: while i ≥ 0 do
 9:     l ← s[0 : i]
10:     r ← s[i + 1 : length(s) − 1]
11:     if l ∈ d ∧ ¬isPrefix(l) ∧ r ∈ d ∧ ¬isSuffix(r) then
12:         return l + " " + r
13:     else if l ∈ d ∧ ¬isPrefix(l) then
14:         r ← BackwardSplit(r, d)
15:         if r was further split then
16:             return l + " " + r
17:         end if
18:     end if
19:     i ← i + 1
20: end while
```

To overcome the shortcomings of Greedy, we propose an alternative algorithm (Backward) that walks through the hard word from the end to the beginning. Algorithm 2 shows the pseudocode of the Backward algorithm. As before, there is no need to split a soft word, a single character, or an empty string (line 4-6). Otherwise, it loops through all the possible split positions in s from the end to the beginning. If both the left (l) and the right (r) substrings with respect to the current split position are soft words (but cannot be common prefixes and suffixes), then the split is made affirmatively (line 11-12). However, if only l

is a soft word, the algorithm is called recursively on r. The split is committed only if r was further split (13-18). Thus, *casesensitive* is correctly split into *case sensitive* since *sensitive* cannot be further split while both *case* and *sensitive* are soft words. The algorithm also successfully avoids splitting identifiers such as *browserid* into *brow ser id*.

Our last algorithm uses dynamic programming to split identifiers. Dynamic programming is good at finding global optimal solutions for optimization problems [50]. Thus, the identifier splitting problem can be transformed into the optimization problem where the goal becomes finding a split that maximizes the length of the longest word that exists in the dictionary. Suppose we have a same-case identifier s with n letters represented as $\{s_1, \ldots, s_n\}$ and a dictionary d. Let us define a table $T[n, k]$ to record the maximum length of the longest substring of all possible splits of $\{s_1, \ldots, s_n\}$ into k ranges. A substring has length of zero if it does not exist in d. Thus, we use the following recurrence relation to compute the values for the table:

$$T[n,k] = \max_{i=1}^{n-1}\left\{ max\left(T[i,k-1], \text{length}(\{s_{i+1}, \ldots, s_n\}) \right) \right\}$$

The initial conditions of the recurrence relation to initialize the first row and the first column are:

$$T[1,i] = \text{length}(\{s_1\}) \text{ for all } 1 \leq i \leq n$$

$$T[j,1] = \text{length}(\{s_1, \ldots, s_j\}) \text{ for all } 1 \leq j \leq n$$

Intuitively, when splitting a single letter s_1 into i ranges, the length of the longest substring in d must either be 0 ($\{s_1\} \notin d$) or 1 ($\{s_1\} \in d$), while when splitting a prefix substring with j letters, the length of the longest substring in d must be either 0 ($\{s_1, \ldots, s_j\} \notin d$) or j ($\{s_1, \ldots, s_j\} \in d$).

In order to reconstruct the optimal split that maximizes the longest substring exists in d,

an additional table D is built to keep track of the positions of the dividers (spaces) that have been inserted into s. Let us define a table $D[n, k]$ to record the index of the last inserted divider when splitting string $\{s_1, \ldots, s_n\}$ into k ranges. We start with the value in $D[n, k]$ and backtrack to get indices for all the dividers.

Note that DP may produce multiple optimal solutions. For instance, DP generates both *on error* and *o n error* as the optimal splits for *onerror*, since both splits acknowledge *error* as the longest substring that exists in d. In situations where there are multiple optimal solutions, we choose the one that minimizes the number of substrings that do not exist in d. Thus, *on error* becomes the final split since all of its substrings (*on* and *error*) exist in d.

4.1.4 Query Parsing

Once we have our preferences split, we can parse the user queries to extract a set of relevant *keywords*. Since we are expecting our queries to be run against identifier-like names, we have adopted a set of rules that limit what keywords that are extracted. The first step removes words with leading numbers, special symbols and punctuation, and converts all of the letters to lowercase. After this step, the user query from our example becomes *firefox doesnt warn me when closing multiple tabs any more*. We filter *stop words* prior to further processing, using a stop words list. We also remove contractions using a modified version from [21]. We added words to this list such as *default*, *enable*, and *disable* because they are generic and carry a little discriminating power when it comes to configurations. The above query thus becomes *firefox warn closing multiple tabs*, which only contains the keywords that carry the core information.

The size of the user query has been reduced from the previous steps now without losing the core information. Preferences that contain any of these words should be considered as relevant to the user. However, the query may fail to match the desired preferences if the

user expresses the same concept using slightly different words that have similar meaning, rather than using the exact words in the preference names or using the same word, but in different word tenses. Assume that the user types the word *closing* to describe the event of closing Firefox. However, preference names are often made up of *root words* (words such as *close*). In addition, some users may use the word *shutdown* instead of *closing*. To alleviate this shortcoming, PrefFinder allows for inclusion of additional lexical databases. In our implementation we evaluate WordNet [32], a lexical database for English, that expands the keywords in a user query with their synonyms and also removes/adds plurals by converting to their root forms. In our running example, WordNet expands this back to 18 keywords with additions such as *shutdown, shutting, closedown, closing, closure, completion, tab*.

4.1.5 Ranker

Once we have parsed both the preferences and the query, the next step is to suggest preferences that are most relevant to the user query. This is a matching problem that is very similar to web searches, where a web search engine searches for web documents that are most relevant to the user query. The difference is that we view the user query and each preference name as *a bag of words* [29], where the order of words does not matter.

To compute the similarity for each (query, preference) pair, we adopt the classic information retrieval weighting scheme *term frequency-inverse document frequency* (*tf-idf*) [3, 29, 48], which measures the importance of a word to a document. Terminologies definitions can be found in Chapter 2. We leave the refinement of this weighting for future work.

On top of the traditional *tf-idf* weight, we impose an additional scale factor which reduces the the effect of synonyms, by scaling down their weight. Our matching favors the term that is found in the original user query. We experimented with a series of scale

factors on the Firefox preference set and found that 0.4 works best as the scale factor for synonyms. Thus, the overall similarity score for a (query, document) pair is computed as the sum of *tf-idf* weights for all the items that occur in both the query *q* and the document *d* by the following equation:

$$\text{score}(q, d) = \sum_{t \in q} tf\text{-}idf_{t,d} \times scale,$$

where $scale$ equals to 0.4 for synonyms, and 1 otherwise.

Table 4.1: Ranking the terms in the correct preference for our example query

item in d	*tf*	*df*	*idf*	*tf-idf*	*scale*	weight
browser	0	300	0.79	0	1	0
tabs	2	30	1.80	3.60	1	3.60
warn	1	21	1.94	1.94	1	1.94
on	0	72	1.41	0	1	0
close	1	13	2.15	2.15	0.4	0.86
other	0	4	2.67	0	1	0

Consider the previous example, where the bag of words after parsing (without the synonyms) are {*firefox, warm, closing, multiple, tabs*} for the query *q* and {*browser, tabs, warn, on, close, other*} for the corresponding preference *d* (*browser.tabs.warnOnCloseOtherTabs*). Table 4.1 shows the statistics of each term in *d* (the preference). The overall score is the sum of the weights of all the terms ($0 + 3.60 + 1.94 + 0 + 0.86 + 0 = 6.41$). Note that term *close* in *d* is a synonym of term *closing* in *q*, and thus is multiplied by a scaling factor of 0.4. The term *browser* has a very low *idf* because it occurs in a large number of preferences (300). The term *tabs* carries more than half of the weight because it matches twice and occurs in only a few preferences (30).

After assigning each preference a similarity score for a given query, all preferences are ranked in decreasing order with respect to the score. The top *n* preferences are written to a rank file (*n* is a parameter specified by the user via PrefFinder front-end UI), which is

subsequently sent to the front-end and displayed.

4.2 Case Study

In this section, we present a case study to evaluate the potential usefulness of PrefFinder. We begin by evaluating the different variations of splitting algorithms, since these are the core to making PrefFinder work. We then use the results from the first question, to inform our evaluation of the overall feasibility of PrefFinder. We end with a comparison of PrefFinder against simple web queries which we believe would be the current state of the art for solving this problem. We answer the following two research questions in this study:
RQ1: How do the identifier splitting algorithms differ in terms of accuracy and efficiency?
RQ2: How effective is PrefFinder in extracting preference options for a set of user queries?
The rest of this section describes our objects of analysis, metrics, and methodology. Artifacts, such as the queries, dictionaries and results can be found on our artifact website[1],

4.2.1 Object of Analysis

Our study object is the open source web browser Firefox version 20.0 on the Ubuntu operating system. In this version, there are 1837 default preferences that can be modified via the `about:config` utility. We implemented both a command-line and interactive version of PrefFinder. The interactive implementation is an extension to the browser. We utilize the Firefox built-in XPCOM API [34] to get direct access to the preference system at run time. We use the XUL [35] markup language for the interface. We use WordNet 3.0, for the synonyms, the iSpell dictionary, a stop words dictionary and a dictionary of computer acronyms. These are available on our associated website.

[1] `http://cse.unl.edu/~myra/artifacts/PrefFinder_2013/`

4.2.2 Study Setup and Method: RQ1

To evaluate our first research question, we focus on the splitting algorithms which operate on the set of preferences. We set up our experiment to determine how well each variant of the algorithm works on the full preference set in Firefox. Since preference names are composed of hierarchical names (split by a period), we first do a trivial pass to get the *identifiers* for each preference. Each identifier can be a single soft word (e.g. *browser*), multiple soft words (*newtab* or *WarnOnClose*). We found 22 identifiers that begin with numbers such as *3208198ce6fd*} or *4447*. We removed these from our experiments. As a result, we have a total of 1594 distinct identifiers that make up the 1837 preference names.

To obtain an oracle for this research question, we asked a research programmer who is not associated with this project (and who has no knowledge of PrefFinder or what we plan to do with this data) to manually split identifiers, so that the resulting words make the most sense from a programmer's perspective. We examined these after the fact, to ensure that the splits looked reasonable. Except in a few cases, where user preference might impact the result (e.g. *name 1* vs. *name1*, we found that we agreed with these splits. Of the 1594 identifiers, 567 identifiers were split during this process into more than one word. The remaining 1027 identifiers were left intact. Table 4.2 summarizes these attributes of our oracle data.

Table 4.2: Preferences and identifier oracle for Firefox 20.0

Number of System Prefs	Number of Identifiers			
	Intact	Split	Distinct	Discarded
1837	1027	567	1594	22

To compare against the oracle, for each of the identifiers, we first run the camel case identifier splitting algorithm. Then we run these identifiers through the three additional splitting algorithms (Greedy, Backward, and DP). The results generated by each algorithm

are compared against the manually produced oracle for an exact match. The effectiveness of each algorithm is measured as the percentage of splits that match oracle exactly.

To evaluate efficiency, we prototype each identifier splitting algorithm separately in Python, and run on a Linux laptop with 2.40 GHz quad-core processor and 6.0 GB RAM. The efficiency of the algorithms is measured as time in seconds to complete the splitting for all of the 1594 distinct identifiers. We run our timing experiments five times and report averages and standard deviations.

4.2.3 Study Setup and Method: RQ2

In order to simulate how users or developers interact with PrefFinder, we collected a set of questions asked by real Firefox users on the Firefox Support Forum [33]. Since people ask many types of questions on this forum, we constrained our choices to those that are related to preferences. We did this by searching on `about:config`. On the date of our query (April 5, 2013), a total of 794 posts were returned as this result. We selected the queries that appear to refer to questions on how to customize Firefox, instead of troubleshooting. For example, ”How do I enable pinch to zoom?” was selected, because it asks about customizing Firefox. However, ”Searching from the url address window has stopped working” was not, because it is considered as a troubleshooting post. An additional requirement, is that the query had to have at least one preference proposed in a follow up post as the solution, since we need an objective oracle. We did a sanity check to validate the proposed solutions (but did not attempt to confirm that they are all correct).

The most commonly used preference solutions, such as *browser.startup.homepage*, were intuitively verified based on our experience. A handful of solutions that fell out of this category were verified by experimentation, such as toggling preference *browser.tabs.onTop* on and off manually to validate that the tab position changes. Some solutions, though,

such as *browser.cache.offline.capacity*, were not verifiable by this approach. Under these situations, we just checked to ensure that these solutions actually exist in the Firefox preference system. In some cases, the follow-up page provided several preferences to answer the user's question. If they all seem reasonable, then we allow all to serve as the oracle and we use the first match on any of the possible solutions. In our final set of questions, 20% had two or more possible answers. We stopped when we had 100 queries that matched our criteria.

To keep the experiment as realistic as possible, queries are collected exactly as they appear in the posts without any formatting by using a copy and paste. As a result, queries retain their original formats with punctuation, special symbols, white space, mixed-case letters, etc. Table 4.3 shows a subset of our queries along with the oracle. Query number 5 has four possible answers as its oracle. The full set of queries with links to the original postings can be found on our website.

We then ran each query through PrefFinder. Preferences were returned in ranked (descending) order. We measure the effectiveness of PrefFinder by counting the number of (non-zero ranked) returned preferences (the smaller number of returned preferences is better because this is less work for our user), and the rank position in which the (first) correct solution is returned (or not found if the solution is not within the returned set).

As a second part of this study, we compare PrefFinder against regular web queries. Since we don't have another tool to compare PrefFinder against, we think that this is the best option. We evaluate the ranking positions and simulated human cost (measured as the number of page screens that a user must go through to find the answer). To set up this part of the study, we use Google on a subset of the user queries. Before each web search, we clear the entire browsing history, cache, and cookies of the browser to ensure that the current search is not affected by previous searches, and we copy-and-paste the exact user queries into Google. Note that some queries may not contain the keyword *firefox*. To ensure

Table 4.3: Sample of queries from the Firefox help forum

Query	**Preference**
(1) How to change permanently the Search Engine?	keyword.URL
(2) Is there an about:config entry to toggle Search Example.com for selected text automatically switching to the tab it opens?	browser.search.context.loadInBackground
(3) Does this signify compromised https pages that I visit?	network.websocket.allowInsecureFromHTTPS
(4) How do I prevent the warning for closing multiple tabs at once from displaying?	browser.tabs.warnOnCloseOtherTabs
(5) How do I enable pinch to zoom?	(a) browser.gesture.pinch.in, (b) browser.gesture.pinch.in.shift (c) browser.gesture.pinch.out, (d) browser.gesture.pinch.out.shift
(6) disable telemetry prompt	toolkit.telemetry.prompted

a fair comparison, we append keyword *firefox* to such queries to make them relevant to our object of analysis. We discard the web pages where we obtained our original solution.

4.2.4 Threats to Validity

We describe the main threat to validity of our experiments. First, we built our PrefFinder just on two systems (Firefox and LibreOffice). But we used real user queries and we believe that this is a representative highly configurable system for which PrefFinder would be useful. We need to examine more wide-ranged systems to justify that our approach will generalize. Second, some of the preference solutions proposed by the forum follow-up posts are not verifiable; we took them as the ground truth. In addition, we acknowledge that there exist other sophisticated NLP and IR techniques in the literature that may produce better splitting and ranking results, that we did not explore. For our web search results, we removed the website on which we obtained the original query oracle. This biases the results, but since we obtained these from the Firefox website, we thought that would be an unfair advantage for web search and assume that this tool would be used when such a utility such as the Firefox help forum is not immediately available (i.e. when a user is offline).

4.3 Results

In this section, we first present the comparison results of different identifier splitting algorithms, and then discuss the PrefFinder results.

4.3.1 RQ1 Identifier Splitting

To answer RQ1, we analyze different identifier splitting algorithms separately. As discussed in section 4.1.3, our identifier splitting algorithm has two steps. In the first step, it splits the camel case identifiers with the Camel Case splitting algorithm. In the second

step, it passes the resulting identifiers to one of the same case splitting algorithms (Greedy, Backward, DP) for further splits. We analyze each step individually.

Table 4.4 summarizes five scenarios to illustrate the impact of each algorithm. The first row contains the original identifiers retrieved from Firefox preference names and the second row contains the manual oracle corresponding to each identifier. The following rows show the splitting results by each algorithm . In the first column example, the camel case detects camel case letters in the original identifier *SOAPHeaderBlock*, and thus makes the correct splits. Since the resulting identifiers *SOAP*, *Header*, and *Block* are soft words, none of the same case splitting algorithms have any effect. As a result, camel case is sufficient for this example. The second example is one in which the camel case algorithm correctly splits the word, and all three of the other algorithms splits this further into an incorrect answer. The word *Sidebar* was left as a single word by our human oracle, who deems this a valid programming word. This type of false negative, may be overcome with good synonym databases in the search phase. The other three examples have mixed results. In the case of *printsettings*, the camel case algorithm fails (this is expected), but the other algorithms, except for greedy found the correct split. The greedy algorithm works forward to find the longest word, so it made its first split on *prints*. In the case of *pagethumbnails*, only the backward algorithm provides the correct split. It looks for the longest word starting at the end of the word. Finally in the last example *composer2d*, both the greedy and DP approach are correct, while the backwards algorithm fails.

Table 4.5, presents the overall results of our splitting. The first two columns show the number and percentage of correct splits made on the 567 identifiers that the oracle deemed should be split. Camel Case splits correctly 67.6% of the time. However, many identifiers require additional splits. As can be seen, the other algorithms have higher correctness percentages. The backward algorithm, is correct 88.9% of the time, while the next best is the Greedy at 83.8% of the time. The last two columns show the number and percentages

of *false splits* made on the 1027 identifiers that our oracle deemed were not splittable. Camel case makes the smallest number of false splits (0.4%). The Backward algorithm makes only 6.7%, while the algorithm that had the highest number of false splits is the DP (16.2%). Table 4.6 shows the percentage of correct splits if we use all of the 1594 identifiers together. As can be seen, Backward has the highest percentage overall (91.7%). Interestingly, Camel Case (88.2%) outperforms both Greedy (86.4%) and DP (83.8%) when we examine all of the data.

Table4.7 shows the five run times for each algorithm across all of the 1594 distinct identifiers. As can be seen, running Camel Case alone only takes 0.04 seconds to complete on average. The execution time for Greedy (9.17s) and Backward (9.31s) are about the same with small standard deviations. However, DP takes 231.531s to finish on average, which is about 25 times slower than Greedy and Backward. This is because for each identifier, DP builds two $n \times n$ tables (T and D) to keep track of the recurrence relations and the divider positions. It is possible that we can improve this slightly through a more efficient implementation, however the runtime complexity of the algorithm means that it will not be as completive as the other two.

RQ1 Summary We summarize our findings for RQ1 by concluding that on our prototype system, the combination of Camel Case with the Backward split is the most effective approach. Adding the Backward split incurs some additional runtime, but it is small. The most expensive algorithm and worst performer is the DP. Camel Case does surprisingly well overall, since it doesn't suffer from a high number of false splits which occur in both the Greedy and DP algorithms. We use the Backward algorithm to answer our next research question.

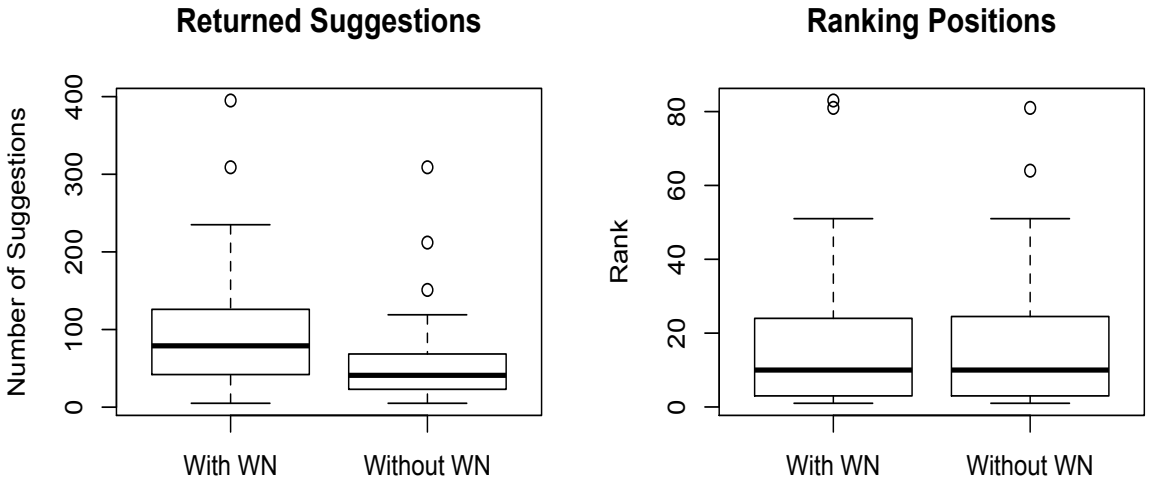

Figure 4.3: Total number of returned suggestions (left) and the associated ranking positions (right) for the successful queries

4.3.2 RQ2 PrefFinder Suggestions

To answer RQ2, we chose to examine two scenarios. One group uses WordNet to expand the set of words in the user query with synonyms, while the other group has WordNet turned off. PrefFinder found the correct solution for 73 queries in the WordNet group and 72 when WordNet was not used. We show the data from this question in two box plots (Figure 4.3).

The left graph shows data on the number of returned suggestions for each of the queries that was correctly found (the overall data which includes the failed queries is similar). As can be seen there is a wide range of values for both groups. Some queries have only a few suggestions, while a few have as many as several hundred responses. However, the overall trend shows that the trend is towards smaller numbers with the medians under 100. It is interesting to see that in the WordNet group (left plot in each graph), there are more suggestions returned. Since the synonyms expand the set of keywords this increases the probability of more solutions having non-negative rankings. However, it also means that PrefFinder is likely to match other irrelevant results as well which introduces noise.

Thus, we further examine the ranking positions of working solutions to analyze the impact of noise. The box plot on the right shows the ranking positions for both sets of queries. There is no discernible difference between the group with and without WordNet (with medians around 10 and outliers as high as 80). Considering that there are almost 1900 preferences, we believe that this is a reduction in the work a user would need to perform without PrefFinder. As shown at the bottom of Table 4.7, it takes WordNet 0.31s on average to extract synonyms for all 100 user queries, which is about 3.1ms for each query.

We examine the success rates of our queries further by using adopting the *top 10* cut-off point criteria for web searches from [9]. This states that a search is successful if the system finds the answer within the top 10 entries (denoted as S@10). This claim is based on the fact that the top 10 search results typically appear on the first page of a web search and users are likely to look through only the top 10 entries before issuing a new query. Figure 4.4 presents the number of queries, for which PrefFinder is able to find a working solution, for various criteria. As we can see, when only considering the top 10 search results (S@10), PrefFinder successfully finds solutions for 40 user queries in the WordNet group and 38 queries for the group without WordNet. About 70% of the solutions are found within the top 50 (S@50) for both groups. Here we see that WordNet does slightly worse at the lower ranks, but we do not believe it is significant.

We next examine more closely a subset of the queries against a web search (as described in the study). We chose to examine the 27 queries that we found with WordNet within the top 5 choices. As mentioned, we ignored the oracle webpage (which is usually the first page to be returned in our search). We manually examined each page and counted the number of screens a user has to scroll through to read that whole page (measured as the number of spacebars it takes to reach the end of the page). Figure 4.5 shows the ranking positions of the PrefFinder queries versus the web queries. PrefFinder finds more working solutions than the web searches for all the ranking positions except the top position, where

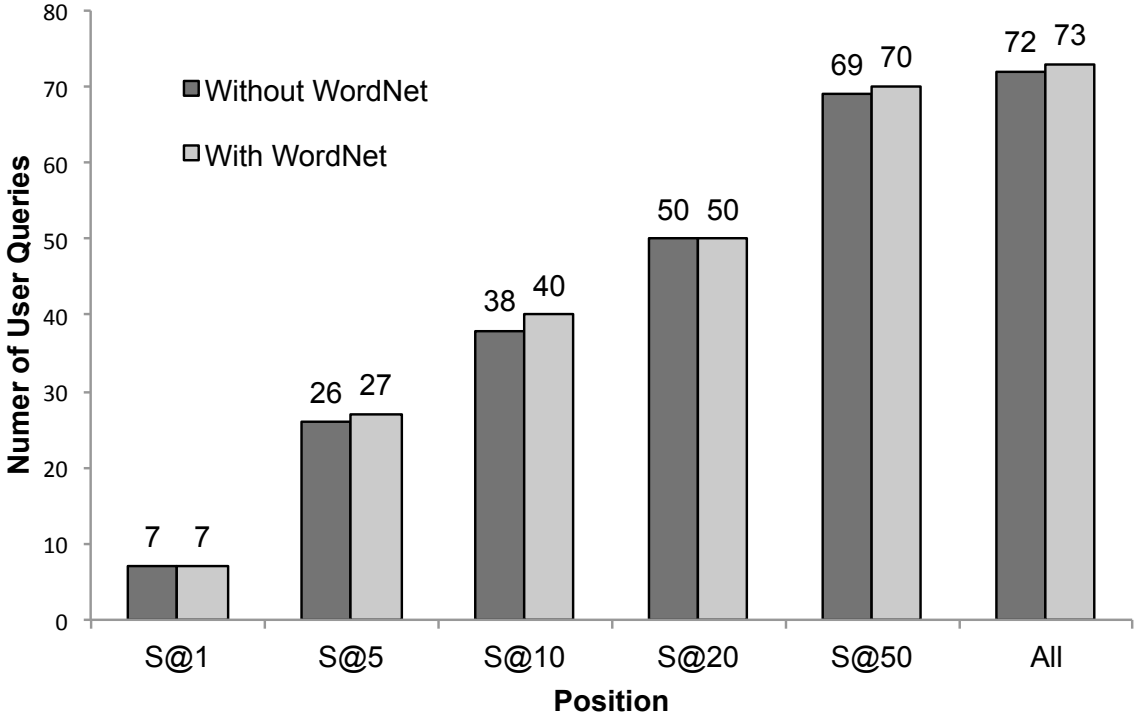

Figure 4.4: Rank positions for successful queries

web searches find 15 working solutions and PrefFinder finds 7. There are also 7 queries where the web search failed to find the query within the top 5 results. When we average the number of pages that a user has to potentially examine (versus only a single preference name), the web searches have an average of 8.2 screens for this set of queries. In addition the web pages may have links (we assume that the user does not leave the page). From this small study we conclude that PrefFinder is "competitive". Clearly if a forum such as the Firefox forum exists and the user has online access, this is a viable option, but they will have to sort through a large amount of text to find the answer. Future work will evaluate the human aspects of PrefFinder.

RQ2 Summary We conclude from this data that PrefFinder has potential to extract the correct preferences for real user queries. The main challenge moving forward will be to improve our ranking and splitting algorithms and to improve the ranking of correct so-

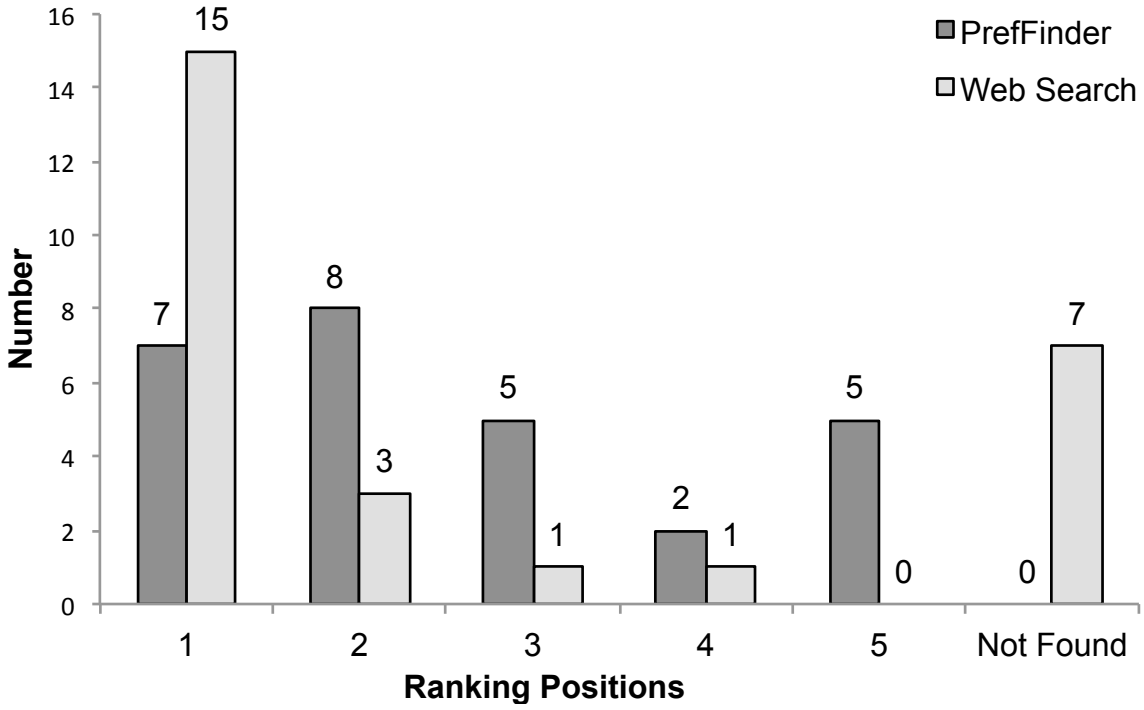

Figure 4.5: PrefFinder vs. a web query

lutions. Given that there is little time overhead and that the location of the solution in the rankings is the same, and that WordNet returned one additional result, we don't believe that this hurts PrefFinder, but a deeper analysis is needed to determine when it will be beneficial to use.

4.4 Summary

In this chapter we have presented PrefFinder, a natural language based querying framework to recommend and customize configurable options. We have evaluated PrefFinder and several variants of our parsing algorithms to improve matches in this context. Using 100 queries obtained from an online forum, we determine that using a backward search during word splitting, combined with a synonym database, achieves the best results. The correct configuration option is found 50 percent of the time within the top 20 choices, and 73

percent of the time overall. In a comparison against a standard web search, we show that PrefFinder is competitive in finding the answer, but at a potentially lower cost.

Table 4.4: Examples of the results of the different splitting algorithms

Identifier	SOAPHeaderBlock		active Sidebar		printsettings		pagethumbnails		composer2d	
Oracle	SOAP Header Block		active Sidebar		print settings		page thumbnails		composer 2d	
Camel Case	SOAP Header Block	✓	active Sidebar	✓	printsettings	×	pagethumbnails	×	composer2d	×
Greedy	SOAP Header Block	✓	active Side bar	×	prints ett in gs	×	page thumbnail s	×	composer 2d	✓
Backward	SOAP Header Block	✓	active Side bar	×	print settings	✓	page thumbnails	✓	composer2d	×
DP	SOAP Header Block	✓	active Side bar	×	print settings	✓	page thumbnail s	×	composer 2d	✓

Table 4.5: Results of splitting on the 567 identifiers which should be split

Algorithm	Correct	Pct.(%)	False Splits	Pct.(%)
Camel Case	383/567	67.6	4/1027	0.4
Greedy	482/567	85.0	132/1027	12.9
Backward	504/567	88.9	69/1027	6.7
DP	475/567	83.8	166/1027	16.2

Table 4.6: Comparing splitting quality against the human oracle on all distinct identifiers

Algorithm	Correct	Pct.(%)
Camel Case	1406/1594	88.2%
Greedy	1377/1594	86.4%
Backward	1462/1594	91.7%
DP	1336/1594	83.8%

Table 4.7: Time to split 1594 distinct identifiers (top) and to extract synonyms from WordNet for 100 user queries (bottom)

Run	1	2	3	4	5	Avg.	Std.
Time to split 1594 distinct identifiers (sec)							
Camel Case	0.05	0.04	0.04	0.04	0.04	0.04	0.00
Greedy	10.01	8.97	8.98	8.94	8.94	9.17	0.47
Backward	10.09	9.13	9.15	9.10	9.10	9.31	0.43
DP	248.17	232.93	226.62	225.11	224.82	231.53	9.86
Time to extract synonyms from WordNet for words in 100 user queries (sec)							
WordNet	0.19	0.31	0.41	0.27	0.37	0.31	0.09

Chapter 5

Conclusions and Future Work

In this thesis we have presented an analysis of configurability in real world software systems to evaluate the complexity that configurability adds for developers and testers and built a natural language based querying framework, PrefFinder, to identify configurable options.

For the analysis, we have studied three highly-configurable software systems. We have shown that our open source and industrial applications all have similar mechanisms for maintaining and modifying configuration options and presented an abstraction of this mechanism. We also see that there is no single (easily available) ground truth to determine the full possible configuration space. To this end we recommend merging multiple sources, developing cross-language analysis tools and providing traceability between the different configuration layers. We have also seen that the dynamic behavior can be difficult to understand, therefore we need to be cognizant of the lifecycle of the application to understand our exact configuration state during debugging. In order to address preference recommendation and customization issues for large scale highly-configurable software systems, we developed PrefFinder, which uses several splitting algorithms informed by databases of stopwords, and dictionaries of synonyms. We evaluated PrefFinder using only camel

case splitting, and with several additional splitting algorithms. Our best results, first use camel case followed by a backward splitting algorithm. In our analysis of a set of 100 user queries, PrefFinder found the oracle solution within the top 10 choices, 40 percent of the time, within the top 20 choices percent of the time, and 73 percent overall. When compared with a web search we show that PrefFinder is competitive in finding the answer, at a potentially lower cost.

In future work we plan to implement some configuration merging techniques, and traceability links between the various layers. We also plan to examine a larger variety of highly-configurable systems with larger evaluation to understand if the same model holds. In addition, we plan to refine the algorithms for splitting and ranking to improve our overall matching. We will also evaluate the ability to automatically set the values of options once discovered and to perform human studies. Finally, we plan to connect PrefFinder with some existing analysis tools and to build prototypes for additional configurable systems such as LibreOffice.

Bibliography

[1] Computer acronyms list. `http://www.francesfarmersrevenge.com/stuff/archive/oldnews2/computeracronyms.htm`.

[2] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *International Symposium on Foundations of Software Engineering*, FSE, pages 308–318, 2008.

[3] David Binkley and Dawn Lawrie. Development: Information retrieval applications. In *Encyclopedia of Software Engineering*, pages 231–242. 2010.

[4] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Conference on Computer Supported Cooperative Work*, CSCW, pages 301–310, 2010.

[5] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *International Conference on Software Engineering, ICSE*, pages 261–270, Minneapolis, Minnesota, May 2007.

[6] Jane Cleland-Huang, Jane Huffman Hayes, and J. M. Domel. Model-based traceability. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 6–10, 2009.

[7] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[8] Myra B. Cohen, Joshua Snyder, and Gregg Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Software Engineering Notes*, 31(6):1–9, 2006.

[9] Nick Craswell and David Hawking. Overview of the trec 2004 web track. In *TREC*, 2004.

[10] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and Denys Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *International Conference on Software Maintenance, ICSM*, pages 22–28, Sep 2013.

[11] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *The Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 290–300, 2013.

[12] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *International Conference on Program Comprehension (ICPC)*, pages 11–20, 2011.

[13] LibreOffice. `http://libreoffice.org/`, 2013.

[14] Emine Dumlu, Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Feedback driven adaptive combinatorial testing. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 243–253, 2011.

[15] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *International Working Conference on Mining Software Repositories, MSR*, pages 71–80, 2009.

[16] Henry Feild, David Binkley, and Dawn Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *In Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006*, 2006.

[17] BradyJ. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Failure avoidance in configurable systems through feature locality. In Javier Camára, Rogério Lemos, Carlo Ghezzi, and Antónia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 266–296. Springer Berlin Heidelberg, 2013.

[18] Ispell. `http://www.gnu.org/software/ispell/`.

[19] O. Gotel, J. Cleland-Huang, J. Huffman Hayes, A. Zisman, A. Egyed, P. Grunbacher, and G. Antoniol. The quest for ubiquity: A roadmap for software and systems traceability research. *International Requirements Engineering Conference, RE*, 0:71–80, 2012.

[20] Steffen Herbold, Jens Grabowski, Stephan Waack, and Uwe Bünting. Improved bug reporting and reproduction through non-intrusive GUI usage monitoring and automated replaying. In *International Conference on Software Testing, Verification and Validation Workshops*, ICSTW, pages 232–241, 2011.

[21] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expan-

sions in programs to enhance software maintenance tools. In *International Working Conference on Mining Software Repositories (MSR)*, pages 79–88, 2008.

[22] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based semantically-similar words from comment-code mappings. In *Working Conference on Mining Software Repositories*, may 2013.

[23] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *International Conference on Software Engineering Companion Volume, Software Engineering in Practice*, SEIP, pages 215–224, 2014.

[24] Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In *International Conference on Software Engineering*, ICSE, pages 474–484, 2012.

[25] Luis C. Lamb, Waraporn Jirapanthong, and Andrea Zisman. Formalizing traceability relations for product lines. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 42–45, 2011.

[26] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), September 2007.

[27] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), September 2007.

[28] Jonathan I. Maletic and Michael L. Collard. TQL: A query language to support traceability. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 16–20, 2009.

[29] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[30] Andrian Marcus, Xinrong Xie, and Denys Poshyvanyk. When and how to visualize traceability links? In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE, pages 56–61, 2005.

[31] C. McMillan, M. Grechanik, D., C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, September 2012.

[32] George A. Miller. WordNet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

[33] Firefox support forum. `https://support.mozilla.org/en-US/questions/`.

[34] XPCOM. `https://developer.mozilla.org/en-US/docs/XPCOM`.

[35] XUL. `https://developer.mozilla.org/en-US/docs/XUL`.

[36] Firefox. `http://www.mozilla.org/en-US/firefox/`, 2013.

[37] Ohloh. `http://www.ohloh.net/`, 2013.

[38] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. When and how using structural informa-

tion to improve IR-Based traceability recovery. In *European Conference on Software Maintenance and Reengineering, CSMR*, pages 199–208, 2013.

[39] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 75–85, July 2008.

[40] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *International Symposium On Software Testing and Analysis*, pages 75–86, 2008.

[41] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *International Conference on Software Maintenance, ICSM*, pages 255–264, Oct 2007.

[42] Xiao Qu, Mithun Acharya, and Brian Robinson. Configuration selection using code change impact analysis for regression testing. *International Conference on Software Maintenance, ICSM*, 0:129–138, 2012.

[43] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *International Conference on Automated Software Engineering (ASE)*, pages 193–202, nov 2011.

[44] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *International Conference on Software Engineering (ICSE)*, pages 131–140, may 2011.

[45] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *International Conference on Software Engineering*, ICSE, pages 131–140, 2011.

[46] Ariel Rabkin and Randy H. Katz. Precomputing possible configuration error diagnoses. In *Automated Software Engineering*, pages 193–202, 2011.

[47] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *International Conference on Software Engineering*, ICSE, pages 485–494, 2010.

[48] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management*, pages 513–523, 1988.

[49] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *International Conference on Aspect-oriented Software Development*, pages 212–224, 2007.

[50] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[51] Charles Song, Adam Porter, and Jeffrey S. Foster. iTree: efficiently discovering high-coverage configurations using interaction trees. In *The International Conference on Software Engineering*, ICSE, pages 903–913, 2012.

[52] Kathryn T. Stolee and Sebastian Elbaum. Toward semantic search via SMT solver. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 25:1–25:4, 2012.

[53] The Document Foundation. `http://blog.documentfoundation.org/2011/09/28/`, 2011.

[54] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *International Conference on Software Engineering*, ICSE 2012, pages 58–68, 2012.

[55] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.

[56] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles*, SOSP, pages 159–172, 2011.

[57] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *International Conference on Software Engineering*, ICSE, pages 312–321, 2013.

[58] Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *International Conference on Software Engineering*, ICSE, 2014.

[59] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *International Conference on Software Engineering*, ICSE, pages 1074–1083, 2012.